(6)

A Computational Meta Logic for the Horn Fragment of LF

Carsten Schürmann

December 6, 1995

CMU-CS-95-218

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
Department of Philosophy
Carnegie Mellon University

This work was sponsored by the German Academic Exchange Service (DAAD) through the graduate followship program "HSP II/AUFE" and by NSF Grant CCR-9303383.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the German Government or the U.S. Government.

Carnegie Mellon

School of Computer Science
Office of the Dean
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213-3891

1 December 1995

ATTN: Patricia Mawby
Defense Technical Information Center (DTIC)
Suite 0944
8725 John J. Kingman Road
Fort Belvoir, VA 22060-6218

FAX: 703/767-8032

Dear Patricia:

The Computer Science Technical Report Series, published by the School of Computer Science, has traditionally classified all the publications in each yearly series as (A) Unlimited Distribution. Even with the introduction of electronic access, we have not felt a need to change this policy.

Our policy for electronic dissemination, as noted in our public directories, is:

> The reports contained in these ftpable directories are included by the contributing authors as a mechanism to ensure timely dissemination of scholarly/technical information on a non-commercial basis. Copyright and all rights therein are maintained by the authors, despite the fact they have offered this information electronically. It is understood that all individuals copying this information will adhere to the terms/ constraints invoked by each author's copyright.
>
> Reports may be reproduced, but may not be copied for "commercial" redistribution, republication, or dissemination without the explicit permission of the School of Computer Science at Carnegie Mellon and the authors.

We do not consider DTIC as a "commercial" entity, but rather view you as a very valuable resource for providing access to our documents.

Please feel free to contact me with any questions or concerns regarding these reports.

With kind regards,

[illegible]

Catherine Copetas
Assistant Dean

Tel: 412/268-8525 / Fax: 412/268-5576

Keywords: meta logic, logical frameworks, intuitionistic logic, induction, theorem proving, automated program generation, type theory

Abstract

The logical framework LF is a type theory defined by Harper, Honsell and Plotkin. It is well-suited to serve as a meta language to represent deductive systems. LF and its logic programming implementation Elf are also well-suited to represent meta-theoretic proofs and their computational content, but search for such proofs lies outside the framework. This thesis proposes a computational meta logic (MLF) for the Horn fragment of LF. The Horn fragment is a significant restriction of LF but it is powerful enough to represent non-trivial problems. This thesis demonstrates how MLF can be used for the problem of compiler verification. It also discusses some theoretical properties of MLF.

Contents

Acknowledgements

I would like to thank my advisor Frank Pfenning and my reader Bob Carpenter for their guidance and insight. Sincere thanks to my colleagues and friends in the Department of Philosophy and the School of Computer Science: Wilfried Sieg, Teddy Seidenfeld, Ekkehard Rohwedder, Iliano Cervesato and Alberto Momigliano. Your discussions were very helpful and stimulating.

Special thanks also to Stuart Eisenstadt, Oren Glickman, Emilio Miranda, Thomas Polzin and Geir Waagboe, who contributed in their own special ways to the success of this thesis. Finally I want to thank Molly Bigelow for being there through thick and thin.

I also want to thank my sponsors, the German Academic Exchange Service, and the National Science Foundation for their support.

Chapter 1

Introduction

The logical framework LF is a type theory defined by Harper, Honsell and Plotkin [HHP93]. It is very well-suited to serve as a meta language to represent and reason over deductive systems. LF has been implemented as the logic programming language Elf by Pfenning [Pfe89, Pfe92, Pfe94a]. In recent years LF and Elf became more and more popular. Significant problems have been represented in LF and Elf, for example the Church-Rosser theorem [Pfe99] and a structural cut elimination theorem for classical, intuitionistic and linear logic [Pfe94c, Pfe94b].

Elf is a logic programming language and not an automated theorem proving system. Consequently, it serves the purpose of representing meta theoretical results, but it does not support their derivation. The calculus of constructions [CH88, PM93] and Martin-Löf type theory [ML84, ML84] are type theories different from LF. Based on these type theories, many different proof development systems have been implemented: Coq [C$^+$95], LEGO [LP92, Pol94], Nuprl [C$^+$86], Alf [Mag93, MN94, Mag95], and others. These systems are not designed as programming languages.

The aim of this thesis is to define the computational meta logic MLF for a fragment of LF. MLF is based on the intuitionistic sequent calculus with induction. Intuitionistic logic is nicely presented in [Gal93]. MLF is decorated with proof terms which can be interpreted as programs. In this sense MLF is computational. Differently from other implementations of automated theorem proving systems with induction like Coq [C$^+$95], PVS [ORS92, RSC95], and others, MLF does not generate induction principles. Induction hypothesis can be applied to any term, the well-foundedness of the induction must be proven as a property of the proof term. A case distinction rule allows the elimination of LF types.

When using LF as a meta language a common technique for representing an object language is *higher order abstract syntax.* It allows LF variables to mimic a variable concept of the object language [Pfe92]. Consequently, constructs in the object language depending on free variables are represented as functions in LF.

MLF is a meta logic for the Horn fragment of LF. This restriction disallows MLF to prove the existence of a function object in a functional LF type. Consequently, higher order abstract syntax cannot be used in the object language, MLF is used to reason about: The existence of a construct with free variables is not provable. On the other hand, if higher order abstract syntax is not used, MLF is still powerful enough to prove interesting results — as it is shown by examples in this thesis. The meta logic is full intuitionistic logic, the restriction of the Horn fragment refers only to the underlying type theory.

This thesis is organized as follows: In chapter 2 we introduce a toy problem. We show how this toy problem can be represented in LF and the calculus of inductive constructions. We also show an implementation of it in Elf, and how to use Coq's proof engine to derive some meta theoretical results. In chapter 3, we define MLF in terms of its language and its inference rules. Some theoretical properties of MLF are discussed in chapter 4. In Chapter 5 we revisit the example from chapter 2 and describe the derivation of all presented meta theoretical results in MLF.

Chapter 2

Motivation

Type theory can be used to represent difficult and complex theoretical structures in a uniform way. LF type theory is very well suited to represent deductive systems, other type theories are more expressive. In this chapter we will present the logical framework LF and the the calculus of inductive constructions (CIC). Both type theories are used as the underlying theories for several implementations. Elf is a logic programming language based on LF [Pfe89, Pfe94a, Pfe92] and Coq is a proof assistant based on CIC [C^+95].

Type theories are defined as a set of type inference rules. A signature defines basic constants and their types. Signatures represent a set of constructors, from which more complicated terms can be built. A signature in LF is interpreted as a logic program for Elf. Evaluation of a program corresponds to type-checking. A signature in CIC is interpreted as a collection of objects, lemmas and proofs for Coq.

In the theory of programming languages one area of research is to define adequate notions of semantics for programming languages. The goal is to obtain a better understanding of programming languages and their problems. Languages can be compared by their semantics. Programming languages do not have a unique semantics: Denotational semantics identifies computation with a denotation. Operational semantics explains computation in terms of an underlying machine. Natural semantics arises from rewriting theory: For a given reduction ordering the meaning of a program is its canonical form.

The remainder of this chapter is organized as follows: In the first section of this chapter we introduce a toy programming language and define its natural and operational semantics following ideas of Hannan and Pfenning [HP92]. Then we show that if a program has a certain semantical value with respect to the natural semantics, it has the same semantical value with respect to the operational semantics. The reverse direction can also be shown — the reader may consult [HP92, Pfe92]. We will call this theorem *equivalence theorem.* The toy language, both semantic notions and the proof of the equivalence theorem can be represented in LF and implemented in Elf, as we show in in section 2.2. Similarly it is possible to represent and implement the toy language and both semantical notions in CIC. Unlike the representation of the proof of the equivalence theorem in LF, Coq supports the search for the proof. This is described in section 2.3. Finally, in the last section of this chapter, we summarize our experiences with Elf and Coq.

2.1 An Example

We define now the toy programming language $\mathcal{T}$. The language is essentially the simply typed λ-calculus containing two constructs: λ-abstraction and application. Variables are represented by de Bruijn indices. In the first subsection we introduce language $\mathcal{T}$. In the second subsection we define an evaluation judgement, which defines the *natural semantics* of $\mathcal{T}$. In the third subsection we introduce a simulating machine with which we define the *operational semantics.* In the last subsection we state the equivalence theorem and prove one direction.

2.1.1 A Toy Language

The language $\mathcal{T}$ is based on the simply-typed λ-calculus $\lambda^{\rightarrow}$. Variables are represented by de Bruijn indices: in de Bruijn's original formulation [dB72], variable names are encoded by natural numbers. De Bruijn indices replace variable names. Instead of defining variable names with every λ-abstraction, we implicitly assign natural numbers to each λ-expression. The index 1 then refers to the innermost λ-expression, the index 2 to the λ-expression in which the innermost is embedded:

$$\underbrace{\Lambda\, \overbrace{\Lambda\,(2\;1)}^{\text{level 1}}}_{\text{level 2}}$$

This de Bruijn expression is equivalent to $\lambda x.\ \lambda y.\ (x\ y)$. $\mathcal{T}$ will be represented in Elf and Coq. The indices cannot be represented directly: a potentially infinite number of constants would be necessary. Since Elf does not possess an implicit concept of natural numbers, de Bruijn indices have to be defined as: 1 is a de Bruijn index and $N\uparrow$ is a de Bruijn index if N is a de Bruijn index. The syntax of the language $\mathcal{T}$ has the following form:

$$\text{Modified de Bruijn Expressions:} \quad E \quad ::= \quad 1 \mid E\uparrow \mid \Lambda\, E \mid (E_1\ E_2)$$

Application is defined in the standard formulation of the λ-calculus. Note, that by introducing indices this way, de Bruijn expressions can be formed which do not directly correspond to λ-expressions:

$$\Lambda\,\Lambda\,(1\;1)\uparrow$$

is equivalent with

$$\Lambda\,\Lambda\,(1\uparrow\,1\uparrow)$$

We assume all de Bruijn expressions to be closed, i.e. indices do not refer outside the outermost Λ-abstraction.

In the regular λ-calculus, β-reduction is defined as $((\lambda x.M)\ N)$ reduces to $[N/x](M)$. We avoid defining the notion of substitution. Instead we introduce the notion of environment, which represents variable bindings. An environment is represented as a stack of *values.* It is not enough to take de Bruijn expressions as values: we cannot assume that only unevaluated de Bruijn expressions are bound to variables. Evaluated de Bruijn expressions have to include the environment in which they are evaluated. Otherwise, partially evaluated de Bruijn expressions would loose binding information. Therefore, the definition of environment and the definition of values depend mutually on each other:

$$\begin{array}{rrcl} \text{Environments} & K & ::= & \cdot \mid K;W \\ \text{Values} & W & ::= & \{K, E\} \end{array}$$

Values are also called closures. The β-reduction rule for de Bruijn expressions now has the form: $(\Lambda\ M\ N)$ reduces to $\{\cdot;\{\cdot,N\},M\}$. We call $(\Lambda\ M\ N)$ a β-redex for arbitrary M and N. Given an arbitrary de Bruijn E expression, the β rule can be applied to every subexpression of E which is a β-redex. A de Bruijn expression which does not contain any β-redices is called *normal.*

The β-reduction rule does not define in which order β-redices are resolved. We will use a common evaluation order to derive the natural semantics. This evaluation order is referred to as *eager* evaluation. β-reduction is applied in an outermost leftmost order. The evaluation stops if a de Bruijn expression is evaluated which is not a β-redex. This expression is called *canonical.*

Expressions, environments, and values are the basic components of the language $\mathcal{T}$. In the next subsections we define its natural and operational semantics.

2.1.2 Natural Semantics

We represent the natural semantics of the language $\mathcal{T}$ by an evaluation judgment. The judgment is defined to derive the canonical form of a de Bruijn expression using the eager evaluation order.

Let E be an expression, W a value, and K an environment. The judgment

$$K \vdash E \hookrightarrow W$$

puts K, E and W in relation: In a context K, the expression E has the semantical value W with respect to the natural semantics. The set of inference rules according to the eager evaluation ordering is defined as follows: If the de Bruijn expression is of the form 1, it refers to the top element of the environment, in this case W. We can assume W to be already a canonical element. Therefore it does not have to be evaluated further, but represents the result expression.

$$\frac{}{K;W \vdash 1 \hookrightarrow W}\ \mathsf{ev\_1}$$

If the de Bruijn expression is of the form $E\uparrow$, the top element of the environment is not used any more. It is enough to evaluate E in the new smaller environment K. The result of the evaluation is the canonical form of the expression E.

$$\frac{K \vdash E \hookrightarrow W}{K;W' \vdash E\uparrow \hookrightarrow W}\ \mathsf{ev\_shift}$$

In the case that the de Bruijn expression is a Λ-abstraction, the eager evaluation ordering demands not to apply β reduction to any β redex in the body of the Λ-expression. Therefore the result is the closure of K and $\Lambda\ E$.

$$\frac{}{K \vdash \Lambda\ E \hookrightarrow \{K, \Lambda\ E\}}\ \mathsf{ev\_lam}$$

$$\begin{tikzcd} St \arrow[r, "\mathcal{D} :: St \overset{*}{\Longrightarrow} St'"] & St' \arrow[d, "\pi"] \\ E \arrow[u, "\iota"] & W \end{tikzcd}$$

Figure 2.1: Operational semantics of $\mathcal{T}$

The last rule is the application rule. This rule exhibits the eager evaluation order. To evaluate $(E_1\ E_2)$, first E_1 must be evaluated, and the result is applied to the evaluated value of E_2. Note, that the context K is made available to the first and second premiss. The result of the evaluation is independent of the evaluation order of the first two premisses.

$$\frac{K \vdash E_1 \hookrightarrow \{K', \Lambda\, E_1'\} \qquad K \vdash E_2 \hookrightarrow W_2 \qquad K'; W_2 \vdash E_1' \hookrightarrow W}{K \vdash (E_1\ E_2) \hookrightarrow W}\ \textsf{ev\_app}$$

2.1.3 Operational Semantics

The definition of the operational semantics differs quite a lot from the definition of the natural semantics. The operational semantics of $\mathcal{T}$ is defined in terms of the execution behavior of a simulating machine. The machine we are using for our considerations is a CLS machine [Pfe92], which is a state machine with a special instruction set. Each instruction changes the current state of the machine deterministically. A sequence of states visited during a computation is called a *trace*. The evaluation of a de Bruijn expression is described in 2.1. In a first step the de Bruijn expression is mapped into an initial state St, via an embedding function ι. Using the CLS machine, St is transformed into a final state St'. The judgment which expresses this is defined as $St \overset{*}{\Longrightarrow} St'$. The resulting semantical value is projected into the value W, via a projection function π.

The notion of state must refer to the environment, to the program which is to be executed, and to the result calculated so far. The order of the "execution" of the first two subgoals does not play any rôle in the definition of the natural semantics. This is not the case for the CLS machine. Assume the machine is in state S, the execution of a instruction results in state S'. The execution of the next instruction results in a state S''. It is not true that the machine would end up in the same state S'', if both instructions would have been executed in reverse order.

Therefore it is not enough to represent only the actual environment in the state. Environments at earlier stages of the computation must be accessible, so consequently the history of environments has to be stored in form of a stack. This stack serves as a storage space. Every new subcomputation is provided with a new copy of the actual environment on top of the stack.

$$\text{Environment Stacks} \quad KS \ ::= \ \cdot \mid KS; K$$

A similar argument makes the definition of another notion necessary: The result values of subcomputations should not manipulate the partial result of the actual computation. Therefore result values must also be administered by a stack: The result of a subcomputation is always the top element of the value stack:

$$\text{Value Stacks} \quad S \quad ::= \quad \cdot \mid S; W$$

We will now define the instruction set of the CLS machine: First, expressions can serve as instructions. According to the form of an expression, the single step transitions must be defined. Second, there are special instructions, which are executed to combine the results of the execution of subgoals. In the case of the execution of an application $(E_1\ E_2)$, E_1 is executed to obtain a value W_1, then E_2 to obtain W_2, and, finally, W_1 and W_2 are combined. The instruction which performs this combination is *apply*.

$$\text{Instructions} \quad I \quad ::= \quad E \mid \mathit{apply}$$

A program is defined as a sequence of instructions: *done* stands for the "end of execution" flag.

$$\text{Programs} \quad P \quad ::= \quad \mathit{done} \mid I \& P$$

The notion of state is a triple consisting of an environment stack, a program and a value stack:

$$\text{State} \quad St \quad ::= \quad \langle KS, P, S \rangle$$

Next, the state transition function must be defined. A computation is a trace of one or more single step transitions. Each single step transition describes the state change evoked by one instruction. The single step relation is defined by a new judgement $St \Longrightarrow St'$. The rules for this single step transition are formed as axioms:

If the instruction to be executed is of form 1, the top element of the actual environment has to be returned as a value. Note, that the *actual environment* is the top environment on the environment stack.

$$\frac{}{\langle (KS; (K; W)), 1\&P, S\rangle \Longrightarrow \langle KS, P, (S; W)\rangle}\ \mathsf{s\_1}$$

If the instruction to be executed is of form $E\uparrow$, the top element of the actual environment can be discarded and the E has to be executed.

$$\frac{}{\langle (KS; (K; W')), E\uparrow \&P, S\rangle \Longrightarrow \langle (KS; K), E\&P, S\rangle}\ \mathsf{s\_shift}$$

In case that a Λ-expression has to be executed, the result object is the closure of the actual environment and the Λ-expression $\Lambda\ E$. No further subcomputation is necessary.

$$\frac{}{\langle (KS; K), \Lambda\ E\&P, S\rangle \Longrightarrow \langle KS, P, (S; \{K, \Lambda\ E\})\rangle}\ \mathsf{s\_lam}$$

Application is more complicated, because two subcomputation have to be initiated. The first subcomputation calculates the value of E_1, the second the value of E_2.

$$\frac{}{\langle (KS; K), (E_1\ E_2)\&P, S\rangle \Longrightarrow \langle (KS; K; K), E_1\&E_2\&\mathit{apply}\&P, S\rangle}\ \mathsf{s\_app}$$

The instruction *apply* combines the results of both subcomputations. It assumes that both result objects are the top two elements of the result stack. The top element is assumed to be the value of the parameter, the second element is assumed to be the function. Therefore it must be a Λ-abstraction. To combine both computations K' is made the actual environment by pushing it on top of the environment stack, and then the value of the parameter W is pushed onto the actual environment. Afterwards, the body of the Λ-expression is executed: F.

$$\frac{}{\langle KS, \mathit{apply}\&P, (S; \{K', \Lambda\,(F)\}; W)\rangle \Longrightarrow \langle (KS; (K'; W)), F\&P, S\rangle}\ \textsf{s\_apply}$$

A computation trace lies within the transitive closure of single step transitions: The judgement $St \overset{*}{\Longrightarrow} St'$ expresses the existence of a trace from St to St'. A trace can be empty or formed by a single step followed by another trace:

$$\frac{}{St \overset{*}{\Longrightarrow} St}\ \textsf{m\_id} \qquad \frac{St \Longrightarrow St' \quad St' \overset{*}{\Longrightarrow} St''}{St \overset{*}{\Longrightarrow} St''}\ \textsf{m\_step}$$

Finally a new judgement is introduced which defines the semantical value of a program with respect to the operational semantics: In an environment K, the expression E has the semantic value W:

$$K \vdash E \overset{*}{\Longrightarrow\!\!\!\!\!\Rightarrow} W$$

The inference rule for this judgement is easily motivated: E has to be mapped into a state: This is defined by the function ι in diagram 2.1. A state is defined in which the environment stack contains only one element, the environment K. E is directly interpreted as a program. After executing E, the computation must stop: *done* is the instruction executed after E has been executed. The value stack of the initial state is empty:

$$\iota(E) = \langle(\cdot; K), E\&\mathit{done}, \cdot\rangle$$

After the execution is completed, the final state is expected to be of the form $\langle\cdot, \mathit{done}, (\cdot; W)\rangle$. The π function we introduced in diagram 2.1 is therefore defined as

$$\pi(\langle\cdot, \mathit{done}, (\cdot; W)\rangle) = W$$

The goal of the computation is to reach a final state. The environment stack should be empty, since the execution came to an end. The program should contain only the clause *done*. The result value is expected to be the only element on the value stack.

$$\frac{\langle(\cdot; K), E\&\mathit{done}, \cdot\rangle \overset{*}{\Longrightarrow} \langle\cdot, \mathit{done}, (\cdot; W)\rangle}{K \vdash E \overset{*}{\Longrightarrow\!\!\!\!\!\Rightarrow} W}\ \textsf{c\_run}$$

We introduced the language $\mathcal{T}$ and its natural and operational semantics: In the next subsection we state the equivalence theorem, a de Bruijn expression has the same meaning in both semantics. We give the proof of the necessary direction: If a given expression E has the semantical value W with respect to the natural semantics, then E will have the same semantical value W with respect to the operational semantics.

2.1.4 Equivalence Theorem

In this subsection we show that the operational semantics for $\mathcal{T}$ and the natural semantics for $\mathcal{T}$ are equivalent: Given a context K, an environment E and a value W, the equivalence theorem says that

$$K \vdash E \hookrightarrow W \text{ if and only if } K \vdash E \overset{*}{\Longrightarrow\!\!\!\!\!\!\;\twoheadrightarrow} W$$

This theorem is of interest for two reasons. First, the theorem sets both semantical notions in relation. Second, its proof is representable in LF type theory, and therefore also in the Calculus of Inductive Constructions. Section 2.2.1 and section 2.3.1 show details of this representation.

For the proof of the equivalence theorem, we need a preliminary lemma. This lemma is a generalization of one direction of the the equivalence theorem. It is called subcomputation lemma, because it states that every subcomputation ends with a result value on top of the value stack:

Lemma 2.1 (Subcomputation) *Let K be an environment, E be an expression and W be a value. If $K \vdash E \hookrightarrow W$ then for all Ks environment stack, P program and S value stack*

$$\langle (Ks; K), E\&P, S\rangle \overset{*}{\Longrightarrow} \langle Ks, P, (S; W)\rangle$$

Proof: By induction on $\mathcal{D} :: K \vdash E \hookrightarrow W$.

Case: $\mathcal{D}$ ends in an application of the rule ev_1.

$$\mathcal{D} = \frac{}{(K; W) \vdash 1 \hookrightarrow W}\,\textsf{ev\_1}$$

Hence we have by using s_1

$$\langle (KS; (K; W)), 1\&P, S\rangle \Longrightarrow \langle KS, P, (S; W)\rangle$$

Using rules for the multi step computations, we immediately get a derivation for $\langle (KS; (K; W)), 1\&, S\rangle \overset{*}{\Longrightarrow} \langle KS, P, (S; W)\rangle$.

Case: $\mathcal{D}$ ends in an application of the rule ev_shift:

$$\mathcal{D} = \frac{K \vdash E \hookrightarrow W}{K; W' \vdash E\uparrow \hookrightarrow W}\,\textsf{ev\_shift}$$

Then

$\langle (Ks; (K; W')), E\uparrow \&P, S\rangle$
$\Longrightarrow\ \langle (Ks; K), E\&P, S\rangle$ By s_shift
$\overset{*}{\Longrightarrow}\ \langle Ks, P, S; W\rangle$ By induction hypothesis

Therefore there is a derivation for $\langle (Ks; (K; W')), E\uparrow \&P, S\rangle \overset{*}{\Longrightarrow} \langle Ks, P, S; W\rangle$.

Case: $\mathcal{D}$ ends in an application of the rule **ev_lam**:

$$\mathcal{D} = \frac{}{K \vdash \Lambda\, E \hookrightarrow \{K, \Lambda\, E\}}\ \textsf{ev\_lam}$$

Then $\langle (KS; K), \Lambda\, E \& P, S\rangle \overset{*}{\Longrightarrow} \langle KS, P, (S; \{K, \Lambda\, E\})\rangle$ follows directly from the definition of **s_lam**.

Case: $\mathcal{D}$ ends in an application of the **ev_app** rule.

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1}{K \vdash E_1 \hookrightarrow \{K', \Lambda\, E_1'\}} \qquad \overset{\mathcal{D}_2}{K \vdash E_2 \hookrightarrow W_2} \qquad \overset{\mathcal{D}_3}{K'; W_2 \vdash E_1' \hookrightarrow W}}{K \vdash (E_1\ E_2) \hookrightarrow W}\ \textsf{ev\_app}$$

Then

$$\begin{array}{llr}
 & \langle (KS; K), (E_1\ E_2) \& P, S\rangle & \\
\Longrightarrow & \langle (KS; K; K), E_1 \& E_2 \& \mathit{apply} \& P, S\rangle & \text{By rule } \textsf{ev\_app} \\
\overset{*}{\Longrightarrow} & \langle (KS; K), E_2 \& \mathit{apply} \& P, (S; \{K', \Lambda\, E_1'\})\rangle & \text{By ind. hyp. on } \mathcal{D}_1 \\
\overset{*}{\Longrightarrow} & \langle KS, \mathit{apply} \& P, (S; \{K', \Lambda\, E_1'\}; W_2)\rangle & \text{By ind. hyp. on } \mathcal{D}_2 \\
\Longrightarrow & \langle (KS; (K'; W_2)), E_1' \& P, S\rangle & \text{By rule } \textsf{s\_apply} \\
\overset{*}{\Longrightarrow} & \langle KS, P, (S; W)\rangle & \text{By ind. hyp. on } \mathcal{D}_3.
\end{array}$$

□

Now we can state the equivalence theorem and prove one direction of it. The other direction is omitted, the reader is referred to [Pfe92].

Theorem 2.2 (Equivalence Theorem) *For K environment, E expression and W value:*

$$K \vdash E \hookrightarrow W \quad \textit{if and only if} \quad K \vdash E \overset{*}{\Longrightarrow\!\!\!\!\rightarrow} W$$

Proof: $\Rightarrow$: Apply subcomputation theorem to K,E,W and $K \vdash E \hookrightarrow W$. Choose the environment stack to be $\cdot$, the program to be *done* and the result stack to be $\cdot$:

$$\langle (\cdot; K), E \& \mathit{done}, \cdot\rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot; W)\rangle$$

Apply **c_run** to obtain:

$$K \vdash E \overset{*}{\Longrightarrow\!\!\!\!\rightarrow} W$$

$\Leftarrow$: See [Pfe92]. □

2.2 LF and Elf

Elf is a logic programming language based on LF type theory. Basic theoretical work has been done by Plotkin, Honsell, and Harper [HHP93]. Pfenning implemented LF type theory in form of the logic programming language Elf [Pfe91]. The goal of this section is to introduce LF type theory and to motivate the advantages of using LF type theory as a programming language. We will also demonstrate the need for a meta logic for LF.

This section is divided into two subsections. In the first subsection LF type theory is introduced. In the second subsection the representation of the language $\mathcal{T}$ in LF is given and the implementation of $\mathcal{T}$ in Elf is explained. This presentation will follow closely [Pfe92]. We then will discuss why meta logical reasoning over LF signatures within LF is burdensome: Elf is a programming language and is not anticipated to be a theorem prover.

While this section is concerned with LF and Elf, the next section gives an overview of the calculus of inductive constructions [PM93] as the theoretical foundation of Coq 5.10.[C^+95].

2.2.1 LF Type Theory

LF type theory was introduced by Harper, Honsell and Plotkin in [HHP93]. It can be seen as an extension of the simply typed λ-calculus by introducing dependent types and a rigorous distinction between object level and type level. Note, that this distinction is blurred for CIC as we will see in the next section: there is no syntactical distinction between types and objects.

The strict distinction between *object* level and *type* level has a pleasant side effect. There is a natural way to present deductive systems in LF type theory. Objects can be interpreted as derivations and judgments can be interpreted as types. This is often referred to as the judgment-as-type paradigm. It makes a logic interpretation of LF type theory possible: Types can also be interpreted as propositions, their truth value depends on whether the type is empty or not. If the type is not empty, it is called *inhabited* and the corresponding object corresponds to the proof. The combination of LF type theory, deductive systems, the logic interpretation and special characteristics of the inference rule system enables LF type theory to be used as a foundation for the logic programming language Elf.

A closer look at LF type theory reveals that there is another level besides the object and the type level. This level is called the *kind* level. A *kind* is the "type" of a type. Object level, type level and kind level describe all entities which are expressible in LF type theory. The strict distinction between object level and type level prevent the construction of self referential types: there is no need to introduce type universes for LF type theory.

We introduce now the language of each of these three layers. The object layer is represented using the simply-typed λ-calculus. We use M to denote objects. Objects are always typed. A stands for types. The presence of dependent types makes the notion of kind necessary. Kinds are abbreviated by K. There are two different groups of constants: we differentiate constants introduced on the object level from those introduced on the type level. All constants must be properly typed, that is object constants must have a certain type and type constants must be of a certain kind.

Kind:	K	$::=$	$\text{type} \mid A \to K \mid \Pi x : A.K$
Type:	A	$::=$	$a\ M_1 ... M_n \mid A_1 \to A_2 \mid \Pi x : A_1.A_2$
Object:	M	$::=$	$c \mid x \mid \lambda x : A.M \mid M_1\ M_2$
Signature:	Σ	$::=$	$. \mid \Sigma, c : A \mid \Sigma, a : K$

Objects can be either variables, constants, applications, or λ-abstractions. Since there is the notion of function on the object level, there must be a function type on the type level: the Π-type. There are no type variables in LF, only object variables. Types which are dependent on parameters are called *type families.* In our presentation of LF we define the arrow types as syntactic sugar: The type $A_1 \to A_2$ is a function type from A_1 to A_2. This is only a special case of the Π-type: $\Pi x : A_1.\ A_2$ where x does not occur free in A_2.

Dependent kinds are used to "type" dependent types: Without dependent types there would be only one kind : type. To assign a kind to a function type, dependent kinds have to be introduced: $\Pi x : A.\ K$. It should be clear from the context if a Π-application constructs a type or a kind. As in the case for types, arrow kinds are defined as syntactic sugar. $A \to K$ stands for $\Pi x : A.\ K$, where x does not occur free in K.

Let c, a be a constant names: $c : A$ is called an *object constant declaration* and $a : K$ is called a *type constant declaration.* The *signature* Σ is defined as a list of such declarations. "$\cdot$" stands for the empty signature.

We will now introduce the type system of LF. To define the general typing judgments we have to introduce the notion of a context. A *context* represents a list of variables and their types. Variables are always object variables, so their types must be defined on the type level. "$\cdot$" stands for the empty context. In the following we omit the leading "$\cdot$," for non-empty contexts. Here is the syntactical definition of contexts in LF:

$$\text{Context:}\quad \Gamma \quad ::= \quad . \mid \Gamma, x : A$$

Because of the presence of dependent types, well-formed contexts have to be distinguished from ill-formed contexts. The problem of ill-formed contexts stems form the following observation: The types of the variables defined in the context need not to be closed. Let x be a free variable occuring in the type of a variable declaration. The context is *ill-formed* if x is not declared earlier in the context. If x is declared earlier, then the context is called *well-formed.*

Example 2.3 $\Gamma_w = x : A, z : (B\ x)$ *is a well-formed context,* $\Gamma_i = z : (B\ x)$ *is an ill-formed context.*

We define the judgment $\vdash \Gamma$ ctx to express that the context Γ is well-formed. For the set of inference rules defining this judgment consult [Pfe92].

A similar observation holds for signatures: Object and type constants have to be defined before they can be used. This leads to the distinction between well-formed and ill-formed signatures. The reader may consult [Pfe92]. We omit the treatment of signatures in this presentation.

We define the typing judgments for LF objects, LF types, and LF kinds. Note, that all three judgments depend on contexts: kinds and types may also depend on variables.

Definition 2.4 (LF typing judgments) *We write*

$\vdash \Gamma$ ctx	*for* Γ *being a well-formed context*
$\Gamma \vdash M : A$	*for* M *being of type* A, *in context* Γ
$\Gamma \vdash A : K$	*for* A *being of kind* K, *in context* Γ
$\Gamma \vdash K$ kind	*for* K *being a kind in context* Γ

We are not going to give all typing rules — a complete list is given in [HHP93, Pfe92]. As an example consider the typing rule for application of LF objects: For $(M\ N)$ being well-typed we have to show that M is a function of type $\Pi x : A.\, B$, which domain is equivalent to the type of the parameter object N — A. Since B may depend on x, the type of $(M\ N)$ is B, where all free occurrences of x are replaced by N:

$$\frac{\Gamma \vdash M : \Pi x : A.\, B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : [N/x](B)}\,\textsf{objapp}$$

The next rule shows how application on type level is handled: It is of the same form as **objapp** but Π is now a constructor for kinds:

$$\frac{\Gamma \vdash A : \Pi x : B.\, K \quad \Gamma \vdash M : B}{\Gamma \vdash (A\,M) : [M/x](K)}\,\textsf{typeapp}$$

As a last example we give the rule for Π-formation on the kind level. Note, that there is no application on the kind level. The rule reads as follows: If A is a type in a context Γ and K can be shown to be a kind in the extended context $\Gamma, x : A$, then the free occurrence in K can be bound by the Π-constructor and the resulting construct is a kind.

$$\frac{\Gamma \vdash A : \text{type} \quad \Gamma; x : A \vdash K \text{ kind}}{\Gamma \vdash \Pi x : A.\, K : \text{kind}}\,\textsf{kindpi}$$

We do not present the other rules here. We also omit the presentation of equality rules. The reader is referred to [HHP93].

This completes the short introduction into the underlying ideas of LF and its realization. When using LF as a programming language the signature corresponds to a logic program. Constants are the constructors for proof objects. Types stand for propositions. A query corresponds to ask if a type is inhabited. The execution of a query corresponds therefore to type check a type and to construct a proof term for it. If no proof term can be found, the query is said to fail, otherwise it succeeds. Program execution of program A corresponds therefore to the search of a derivation $\cdot \vdash M : A$.

Higher order abstract syntax

Higher order abstract syntax is a special variable representation technique which follows quite naturally from taking the λ-calculus as a representation language for languages which require a variable concept. This technique stems from the following observation: When representing a variable concept there are two possibilities to be considered: First, the variable concept can be represented directly. Variable names are represented by new constants. Substitution must also be represented. Experience shows, that this approach is rather tedious and it is very difficult to represent a variable concept *correctly.* A second possibility is to use the meta variable concept of the type theory instead of implementing a variable concept from scratch. In this case, terms with free variables are represented as function objects in the type theory. The technique is called *higher order abstract syntax* or HOAS. Object variables are represented as meta variables. The following example makes this more clear:

Example 2.5 *Assume a language with a variable concept to be represented: We denote a variable of this concept with $\hat{x}$ and the LF variable with x. Suppose that the function* s *is already represented in LF. Let f be a function which should be represented in LF.*

$$f(\hat{x}) = (\mathrm{s}\ \hat{x})$$

When using HOAS, f is represented as an LF λ-term:

$$f = \lambda x.\ (\mathrm{s}\ x)$$

The variable concept does not have to be represented in LF.

Variable concepts typically make the notion of substitution necessary. How are substitutions represented when using HOAS — and how are substitution applied when using HOAS? The notion of substitution need not to be represented directly in the LF type theory. Substitution application corresponds to β-reduction. And β-reduction corresponds to application on the LF-object level.

It is clear that mimicking substitution application by LF object application lacks generality. It must be proven, that object level application is general enough, to mimic substitution application. This is expressed by the substitution lemma, which has to be proven for every type family, which makes use of HOAS: Let D be a term, which should be represented as an object in LF type theory. We write $\lceil\cdot\rceil$ for the polymorphic representation function. Let $\hat{M}$ be dependent on some variable $\hat{x}$. Let σ be the substitution of the form $[\hat{N}/\hat{x}]$. The notation $[\hat{N}/\hat{x}](\hat{M})$ is a term, where all occurrences of $\hat{x}$ in $\hat{M}$ are replaced by N. The idea of HOAS is it, to replace object variables by meta variables. Therefore $\lceil\hat{M}\rceil$ is actually a λ-expression in LF, which expects another LF object as argument. The representation of substitution application has to be as follows:

$$\lceil[\hat{N}/\hat{x}]\hat{M}\rceil = (\lceil\hat{M}\rceil\ \lceil\hat{N}\rceil)$$

This property has to hold, otherwise HOAS is not applicable for the language in question. It is shown, that it holds in many examples. It is referred to as the substitution lemma in [Pfe92].

Canonical forms

We saw in the last subsection, that higher order abstract syntax can be used to represent object language variables in LF. We saw that it takes some effort to prove the substitution lemma depending on the type family which makes use of HOAS. We also saw that the representation function establishes an identification between derivations in a deductive system and objects in

LF type theory, and between judgments and LF types. In this subsection we want to elaborate further on the similarities between deductive systems and LF type theory.

In general derivations in a deductive system and objects in LF type theory may not be identified: we cannot expect that every object in LF type theory corresponds to a derivation. Consider the deductive system for the judgment $x \in N$, with two inference rules:

$$\frac{}{\mathrm{z} \in I\!\!N}\,\mathsf{z} \qquad \frac{X \in I\!\!N}{X' \in I\!\!N}\,\mathsf{s}$$

The judgment $X \in I\!\!N$ can be represented in LF as

$$\mathrm{nat} : \mathrm{type}$$

the rules can be represented as the following two declarations:

$$\begin{array}{lcl} \mathrm{z} & : & \mathrm{nat} \\ \mathrm{s} & : & \mathrm{nat} \to \mathrm{nat} \end{array}$$

In this deductive system the derivation of $\mathrm{z}''' \in I\!\!N$ has the form

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\mathrm{z} \in I\!\!N}\,\mathsf{z}}{\mathrm{z}' \in I\!\!N}\,\mathsf{s}}{\mathrm{z}'' \in I\!\!N}\,\mathsf{s}}{\mathrm{z}''' \in I\!\!N}\,\mathsf{s}$$

It is represented in LF using the signature and LF typing rules as

$$\mathrm{s}(\mathrm{s}(\mathrm{s}(\mathrm{z}))) : \mathrm{nat}$$

Obviously $\mathrm{z} : \mathrm{nat}$ has a derivation, namely an instantiation of the axiom rule z. On the other hand the $((\lambda x : \mathrm{nat}.\ x)\ \mathrm{z}) : \mathrm{nat}$ does not correspond to a derivation in the system, but it is well-typed. This example shows, that there are more LF objects then derivations. The solution to this problem is to restrict the set of LF objects to canonical LF objects. We omit the details of how canonical elements are defined, the reader my consult [HHP93, Pfe92]. We define a new judgment: An object M is a canonical object of type A in context Γ:

$$\Gamma \vdash M :_c A$$

For any deductive system, we should show adequacy:

If $\hat{M}$ is a derivation in a deductive system for judgment $\hat{A}$ using possible free syntactical variables in $\hat{\Gamma}$, then $\lceil\hat{\Gamma}\rceil \vdash \lceil\hat{M}\rceil :_c \lceil\hat{A}\rceil$

If $\Gamma \vdash M :_c A$ then there should be a derivation $\hat{M}$ of the judgment $\hat{A}$ and a context of variable declarations $\hat{\Gamma}$, s.t. $\lceil\hat{\Gamma}\rceil = \Gamma$, $\lceil\hat{M}\rceil = M$ and $\lceil\hat{A}\rceil = A$.

Adequacy is to be shown for all types and type families. It guarantees the existence of an isomorphism between deductions and LF objects, between judgments and types. If higher order abstract syntax is used, and the substitution lemma holds for the judgment involving HOAS, the isomorphism postulated in the adequacy theorem is called *compositional.*

For the definition of the system MLF, we will need the notion of atomic types:

Definition 2.6 (Atomic type) *Let A be an LF type. A is called an atomic type iff A is canonical and A is not a Π-abstraction.*

In the next subsection we will introduce the logic programming language Elf, and demonstrate how the example from section 2.1 can be represented in Elf by first representing it in LF type theory, and then interpreting the signature as a logic program.

2.2.2 Elf

In this section we give a very short overview about the logic programming language Elf. Then we describe the representation of the language $\mathcal{T}$ in LF which was introduced in section 2.1. In parallel, we give the implementation in Elf: At first we describe the representation and implementation of the syntax of the programming language $\mathcal{T}$ using de Bruijn indices. Then we represent and implement the notion of natural semantics and the notion of operational semantics in LF. Eventually we show the representation and implementation of the equivalence theorem. This presentation follows closely [HP92].

How to use Elf

The process of programming in Elf proceeds in three stages. At the first state, the problem is formulated in form of a deductive system — as done in section 2.1. At the second stage the deductive system is represented in LF type theory: Adequacy and substitution lemmas are proven at this stage. We omit these theoretical considerations in this presentation and refer the reader to [Pfe92]. The judgment $K \vdash E \hookrightarrow V$ for example is represented as the dependent type (feval $K\ E\ V$). Judgments are written in a very mathematical way, using mathematical symbols.LF types are written in the Roman font. At the third stage LF signatures are finally implemented in Elf. This step is quite straightforward: Programming in Elf corresponds to writing signatures in LF type theory. We write Elf source code in typewriter font: the LF type family (feval $K\ E\ V$) for example is implemented as `(feval K E V)`.

The syntax of Elf corresponds directly to the mathematical notation. The keyword `type` stands for the kind type. Let `A`, `B` be types and `K` a kind. λ-abstractions are expressed using the notation `[x:A]M`. Π-abstractions on the type level are written as `(x:A)B` and on the kind level as `(x:A)K`. We also make use of the arrow notation `A -> B` or `A -> K`, which corresponds to $A \rightarrow B$ and $A \rightarrow K$, respectively.

LF signatures correspond to LF programs. A signature is a list of declarations. Let $c : A$ be an object constant declaration and $a : K$ a type constant declaration. The implementation in Elf is then of the form `c : A.` and `a : K.` For a more detailed description of Elf, consult [Pfe89, Pfe92].

Representation and Implementation of $\mathcal{T}$

We will now present the representation of de Bruijn expressions. A de Bruijn expression does not depend on any other objects. Therefore we declare the type (exp) of kind (type): exp is a type constant. We will also refer to it as *type constructor.*

LF		
exp	:	type

The implementation in Elf is very similar:

Elf		
`exp`	`:`	`type.`

The object constants or *object constructors* of the type exp are defined as follows:

LF			Elf		
1	:	exp	`1`	`:`	`exp.`
$\uparrow$	:	exp $\to$ exp	`^`	`:`	`exp -> exp.`
lam	:	exp $\to$ exp	`lam`	`:`	`exp -> exp.`
app	:	exp $\to$ exp $\to$ exp	`app`	`:`	`exp -> exp -> exp.`

The object constructors 1 and $\uparrow$ define the set of de Bruijn indices: 1 corresponds to index 1, $\uparrow$ corresponds to the successor function. It is of type exp $\to$ exp, because it expects one argument: If N is an index, then $(\uparrow N)$ is also an index. In our presentation we use $\uparrow$ as postfix operator: We write $(N\uparrow)$ instead of $(\uparrow N)$.

Since $\mathcal{T}$ makes use of de Bruijn indices, we do not have to implement an explicit variable concept. λ abstraction expects only one argument, an expression of type (exp). Application expects two arguments: The first argument is the function which is to be applied to the second.

In section 2.1 we introduced the notion of values. A value is the closure of an environment and a de Bruijn expression. Environments depend on values. Both notions are mutually recursive. We will therefore first introduce the new types env and val. The object constructor empty defines the empty environment, ";" is the environment constructor and clo is the constructor to create closures.

LF			Elf		
env	:	type	`env`	`:`	`type.`
val	:	type	`val`	`:`	`type.`
empty	:	env	`empty`	`:`	`env.`
;	:	env $\to$ val $\to$ env	`;`	`:`	`env -> val -> env.`
clo	:	env $\to$ exp $\to$ val	`clo`	`:`	`env -> exp -> val.`

Representation and Implementation of the Natural Semantics

In this paragraph we describe the representation of the natural semantics of the language $\mathcal{T}$. The judgment $K \vdash E \hookrightarrow V$ is represented by the dependent type feval. feval depends on three parameters: the current environment, the expression to be evaluated and the result of the evaluation.

LF	Elf
feval : env $\rightarrow$ exp $\rightarrow$ val $\rightarrow$ type	`feval : env -> exp -> val -> type.`

In section 2.1 we defined four different rules for the evaluation judgment:

$$\frac{}{K;W \vdash 1 \hookrightarrow W}\,\textsf{ev\_1} \qquad \frac{K \vdash E \hookrightarrow W}{K;W' \vdash E\uparrow \hookrightarrow W}\,\textsf{ev\_shift} \qquad \frac{}{K \vdash \Lambda\, E \hookrightarrow \{K, \Lambda\, E\}}\,\textsf{ev\_lam}$$

$$\frac{K \vdash E_1 \hookrightarrow \{K', \Lambda\, E_1'\} \qquad K \vdash E_2 \hookrightarrow W_2 \qquad K'; W_2 \vdash E_1' \hookrightarrow W}{K \vdash (E_1\, E_2) \hookrightarrow W}\,\textsf{ev\_app}$$

Because of the derivations-as-objects paradigm, each rule is represented as an object constructor: The type of each constructor depends on the premisses of the rule and all free variables. The rules **ev_1**, **ev_shift**, **ev_lam** and **ev_app** are represented in LF as follows:

LF		
ev1	:	$\Pi K : \text{env}.\, \Pi W : \text{val}.$ feval $(K;W)\ 1\ W$
ev$\uparrow$	:	$\Pi K : \text{env}.\, \Pi E : \text{exp}.\, \Pi W : \text{val}.\, \Pi W' : \text{val}.$ feval $K\ E\ W$ $\rightarrow$ feval $(K;W')\ E\uparrow\ W$
evlam	:	$\Pi K : \text{env}.\, \Pi E : \text{exp}.$ feval K (lam E) $\{K$ (lam $E)\}$
evapp	:	$\Pi K : \text{env}.\, \Pi E_1 : \text{exp}.\, \Pi K' : \text{env}.\, \Pi E_1' : \text{exp}.\, \Pi E_2 : \text{exp}.\, \Pi W_2 : \text{val}.\, \Pi W : \text{val}.$ feval $K\ E_1$ (clo K' (lam E_1')) $\rightarrow$ feval $K\ E_2\ W_2$ $\rightarrow$ feval $(K';W_2)\ E_1'\ W$ $\rightarrow$ feval K (app $E_1\ E_2$) W

Due to the type reconstruction algorithm which is built into Elf we can omit the Π closures in Elf. The encoding in Elf is much more efficient. Note, that the arrow in the Elf code is reversed for the sake of better readability. The program can be read like a Prolog program.

```
                           Elf
fev_1   : feval (K ; W) 1 W.
fev_^   : feval (K ; W') (F ^) W
           <- feval K F W.
fev_lam : feval K (lam F) (clo K (lam F)).
fev_app : feval K (app F1 F2) W
           <- feval K F1 (clo K' (lam F1'))
           <- feval K F2 W2
           <- feval (K' ; W2) F1' W.
```

Representation and Implementation of the CLS Machine

In section 2.1 we defined the general form of the CLS machine. We described the instruction set and the form of programs. Since a CLS machine is a state transition machine we also defined the notion of state. We will now give the representation of the various constructs in LF and the implementation in Elf.

We have to declare four new types: instructions, programs, environment stacks and states. Recall that instructions, programs and environments contribute to the formulation of a state, which is the basic notion for the CLS machine. The CLS runs by calculating traces of states. The final state represents the operational meaning of a de Bruijn expression, as described in diagram 2.1. The representation of the judgments in LF is straightforward. We obtain four new new type constructors: instruction, program, envstack and state.

LF			Elf		
instruction	:	type	`instruction`	:	`type.`
program	:	type	`program`	:	`type.`
envstack	:	type	`envstack`	:	`type.`
state	:	type	`state`	:	`type.`

Instructions can be formed in two ways: first de Bruijn expressions by themselves are instructions. The function ev is an injective embedding function from the type of expressions into the type of instructions. Second a new instruction has to been introduced which combines subcomputations: apply.

LF			Elf		
ev	:	exp $\to$ instruction	`ev`	:	`exp -> instruction.`
apply	:	instruction	`apply`	:	`instruction.`

A program are defined as a list of instructions. The empty program is represented by done. & is the constructor to build up the list of instructions:

LF			Elf		
done	:	program	`done`	:	`program.`
&	:	instruction $\to$ program $\to$ program	`&`	:	`instruction -> program -> program.`

The notion of environment stack is important, because old environments have to be saved in case subcomputations are started. emptys represents the empty environment stack, ;; the constructor.

LF			Elf		
emptys	:	envstack	`emptys`	:	`envstack.`
;;	:	envstack $\to$ env $\to$ envstack	`;;`	:	`envstack -> env -> envstack.`

This completes the representation of the ingredients for the notion of state. Recall, a state contains the environment stack — a storage facility to store environments, the program — in form of a list of instructions — and a storage facility to store results. We did not define a new type for the result stack, since it is only a stack of values which is already defined as env. The state constructor is called st. Here is the representation in LF and in Elf.

LF	Elf
st : envstack $\rightarrow$ program $\rightarrow$ env $\rightarrow$ state	`st : envstack -> program -> env -> state.`

Representation and Implementation of $\mathcal{T}$'s Operational Semantics

In section 2.1 the notion of computation for a CLS machine was defined. A computation is a trace through the state space, ending in a final state. Traces were introduced as multi step transitions. A multi step transition consists of a sequence of single step transitions. Each single step transition corresponds to the execution of a single instructions. The operational semantics of $\mathcal{T}$ is specified by defining these single step transitions. A single step transition is defined as a relation between two states. We denote this relation by the infix operator $\Rightarrow$. It is represented as:

LF	Elf
$\Rightarrow$: state $\rightarrow$ state $\rightarrow$ type	`=> : state -> state -> type.`

Now we represent the single step transition rules: s_1, s_shift, s_lam, s_app and s_apply. The information, if a single step transition is applicable or not, is stored in the types. Here is the representation of the rules:

LF		
c_1	:	ΠH : envstack. ΠK : env. ΠW : val. ΠP : program. ΠS : env. st $(H;;(K\ ;W))$ ((ev 1)&P) $S \Rightarrow$ st H P $(S;W)$
c_$\uparrow$	:	ΠH : envstack. ΠK : env. $\Pi W'$: val. ΠF : exp. ΠP : program. ΠS : env. st $(H;;(K;W'))$ (ev $(F\uparrow)$&P) $S \Rightarrow$ st $(H;;K)$ (ev F&P) S
c_lam	:	ΠH : envstack. ΠK : env. ΠF : exp. ΠP : program. ΠS : env. st $(H;;K)$ (ev(lam F)&P) $S \Rightarrow$ st H P (S;clo K(lam F))
c_app	:	ΠH : envstack. ΠK : env. ΠF_1 : exp. ΠF_2 : exp. ΠP : program. ΠS : env. st $(H;;K)$ (ev(app F_1 F_2)&P) $S \Rightarrow$ st $(H;;K;;K)$ (ev F_1&ev F_2&apply&P) S
c_apply	:	ΠH : envstack. ΠP : program. ΠS : env. $\Pi K'$: env. $\Pi F_1'$: exp. ΠW_2 : val. st H (apply&P) (S;clo K'(lam F_1');W_2) $\Rightarrow$ st $(H;;(K';W_2))$ (ev F_1'&P) S

These five statements can be much easier represented in Elf. The reason is again Elf's powerful type reconstruction algorithm. Here is the representation in Elf:

Elf	
`c_1`	`: st (H ;; (K ; W)) (ev 1 & P) S`
	`=> st H P (S ; W).`
`c_^`	`: st (H ;; (K ; W')) (ev (F ^) & P) S`
	`=> st (H ;; K) (ev F & P) S.`
`c_lam`	`: st (H ;; K) (ev (lam F) & P) S`
	`=> st H P (S ; clo K (lam F)).`
`c_app`	`: st (H ;; K) (ev (app F1 F2) & P) S`
	`=> st (H ;; K ;; K) (ev F1 & ev F2 & apply & P) S.`
`c_apply`	`: st H (apply & P) (S ; clo K' (lam F1') ; W2)`
	`=> st (H ;; (K' ; W2)) (ev F1' & P) S.`

On top of the single step relation the multi step relation is defined. It is the transitive closure of the single step relation. A multi step is characterized by a start state and an end state. The representation of the multi step function is as follows:

LF	Elf
$\overset{*}{\Rightarrow}$: state $\to$ state $\to$ type	`=>* : state -> state -> type.`

The constructor objects are represented in LF:

LF		
id	:	$\Pi St : \text{state}.$ $St \overset{*}{\Rightarrow} St$
~	:	$\Pi St : \text{state}. \Pi St' : \text{state}. \Pi St'' : \text{state}.$ $St \Rightarrow St'$ $\to St' \overset{*}{\Rightarrow} St''$ $\to St \overset{*}{\Rightarrow} St''$

and implemented in Elf:

Elf	
`id`	`: St =>* St.`
`~`	`: St => St'` `-> St' =>* St''` `-> St =>* St''.`

Next, we represent the operational meaning of a de Bruijn expression: Recall, that a de Bruijn expression E is mapped into the state space via the injection function ι. Then a trace is calculated which ends in a final state. This final state is projected into the space of de Bruijn expressions via the function π to obtain the operational meaning W of E. ceval represents the judgement $K \vdash E \overset{*}{\Longrightarrow} W$, with K environment.

LF	Elf
ceval : env $\to$ exp $\to$ val $\to$ type	`ceval : env -> exp -> val -> type.`

The only constructor for ceval has the form:

LF		
run	:	$\Pi K : \text{env}.\, \Pi E : \text{exp}.\, \Pi W : \text{val}.$ st (emptys;;K) (ev E&done) (empty) $\overset{*}{\Rightarrow}$ st (emptys) (done) (empty;W) $\rightarrow$ ceval $K\ E\ W$.

run is implemented in Elf as:

Elf

```
run :      st (emptys ;; K) (ev E & done) (empty)
       =>* st (emptys) (done) (empty ; W)
    -> ceval K E W.
```

This concludes the representation and implementation of the language $\mathcal{T}$ and its natural and operational semantics. As we will see in the next section, when representing $\mathcal{T}$ in CIC and Coq, the inference machine in Coq supports the user to derive meta theoretical result, for example the equivalence theorem 2.2. We can use LF as a representation mechanism, but Elf does not support the search for meta theoretical results. The representation of those results in LF is often possible, when the proof is done by structural induction. Typically these inductive proofs are done by case distinction over some derivation. The theorem is represented as a type and the different cases of the induction proof as constructors for this type. We show in the remainder of this section, how the theorem 2.2 in section 2.1 can be represented in LF and implemented in Elf.

Representation and Implementation of the Equivalence Theorem

We remarked earlier, that the proof of lemma 2.1 has to be formalized a little more, if it should be representable in a formal system. A more rigorous treatment of "How to append multi step transitions" is necessary. Because of its definition a multi step transition can be extended by prefixing it with a single step transition which ends in the start state of the transition. For the proof of case **ev_app** in lemma 2.1, two multi step transitions have to be concatenated. The next lemma guarantees that this kind of concatenation is always possible, i.e the concatenation of two multi step transitions yields a new multi step transition, as long as they end and start in the same state, respectively.

Lemma 2.7 (append) *For every two traces $T : S \overset{*}{\Rightarrow} S'$ and $T' : S' \overset{*}{\Rightarrow} S''$ there exists a trace $R : S \overset{*}{\Rightarrow} S''$.*

Proof: Easy proof by induction over T:

Case: $T = \text{id}$. Therefore $S' = S$. Choose $R = T'$.

Case: $T = A\,\tilde{}\ T''$, with $A : S \Rightarrow S'''$. By induction hypothesis we have a trace $R' : S''' \overset{*}{\Rightarrow} S''$. Construct $R = A\,\tilde{}\ R'$ and $R : S \overset{*}{\Rightarrow} S''$ satisfy the condition.

□

This proof can easily be represented in LF: First we have to represent the lemma in LF: The lemma is transformed into the type append

LF		
append	:	$\Pi St : \text{state}. \Pi St' : \text{state}. \Pi St'' : \text{state}.$
		$St \overset{*}{\Rightarrow} St'$
		$\rightarrow St' \overset{*}{\Rightarrow} St''$
		$\rightarrow St \overset{*}{\Rightarrow} St''$
		$\rightarrow$ type

which can be represented in Elf:

```
Elf
append : St =>* St'
         -> St' =>* St''
         -> St =>* St''
         -> type.
```

Both cases of the proof are represented as constructors of a dependent type:

LF		
apd_id	:	$\Pi St : \text{state}. \Pi St_1 : \text{state}. \Pi C' : St \overset{*}{\Rightarrow} St.$
		append id C' C'
apd_step	:	$\Pi St : \text{state}. \Pi St_1 : \text{state}. \Pi St_2 : \text{state}.$
		$\Pi C : St \overset{*}{\Rightarrow} St_1. \Pi C' : St_1 \overset{*}{\Rightarrow} St_2. \Pi C'' : St \overset{*}{\Rightarrow} St_2. \Pi St_3 : \text{state}. \Pi R : St_3 \Rightarrow St.$
		append C C' C''
		$\rightarrow$ append $(R\tilde{}\ C)$ C' $(R\tilde{}\ C'')$

We implement these constructors in Elf:

```
Elf
apd_id   : append (id) C' C'.
apd_step : append (R ~ C) C' (R ~ C'')
           <- append C C' C''.
```

As we have seen in section 2.1, the concatenation of two multi step derivations is necessary to prove the subcomputation lemma 2.1, which is necessary to proof the equivalence theorem 2.2. We will now give a representation of the subcomputation lemma 2.1 in LF. The idea is the same as in the append lemma: The lemma is represented as type subcomp and the proof as the object constructors for this type:

LF		
subcomp	:	$\Pi K : \text{env}. \Pi E : \text{exp}. \Pi W : \text{val}. \Pi H : \text{envstack}. \Pi P : \text{program}. \Pi S : \text{env}.$
		feval K E W
		$\rightarrow$ st $(H;;K)$ (ev $E\&P$) $S \overset{*}{\Rightarrow}$ st H P $(S;W)$
		$\rightarrow$ type

Its implementation in Elf is similar:

Elf

```
subcomp : feval K E W
          -> st (H ;; K) (ev E & P) S =>* st H P (S ; W)
          -> type.
```

We will now give the representation of the different cases of the proof of lemma 2.1 in LF:

LF		
sc_1	:	env $\to$ val $\to$ envstack $\to$ program $\to$ env $\to$ subcomp ev1 (c_1˜ id)
sc $\uparrow$	:	ΠK : env. ΠF : exp. ΠW : val. ΠKs : envstack. ΠP : program. ΠK_1 : env. ΠD_1 : feval K F W. ΠC_1 : st $(Ks;;K)$ (ev $F\&P$) $K_1 \overset{*}{\Rightarrow}$ st Ks P $(K_1;W)$. val $\to$ subcomp D_1 C_1 $\to$ subcomp (ev$\uparrow$ D_1)(c_$\uparrow$˜ C_1)
sc_lam	:	env $\to$ exp $\to$ envstack $\to$ program $\to$ env $\to$ subcomp evlam (c_lam˜ id)
sc_app	:	ΠKs : envstack. ΠK : env. ΠF : exp. ΠF_1 : exp. ΠP : program. ΠK_1 : env. ΠK_2 : env. ΠF_2 : exp. ΠW : val. ΠW_1 : val. $\Pi C'$: st $(Ks;;K)$ (ev (app F F_1)$\&P$) $K_1 \overset{*}{\Rightarrow}$ st Ks (apply$\&P$) $(K_1$;clo K_2 (lam F_2);$W)$. ΠC_3 : st $(Ks;;(K_2;W))$ (ev $F_2\&P$) $K_1 \overset{*}{\Rightarrow}$ st Ks P $(K_1;W_1)$. ΠC : st $(Ks;;K)$ (ev (app F F_1)$\&P$) $K_1 \overset{*}{\Rightarrow}$ st Ks P $(K_1;W_1)$. ΠC_1 : st $(Ks;;K;;K)$ (ev $F\&$ev $F_1\&$apply$\&P$) $K_1 \overset{*}{\Rightarrow}$ *st* $(Ks;;K)$ (ev $F_1\&$apply$\&P$) $(K_1$;clo K_2 (lam F_2)). ΠC_2 : st $(Ks;;K)$ (ev $F_1\&$apply$\&P$) $(K_1$;clo K_2 (lam F_2)) $\overset{*}{\Rightarrow}$ st Ks (apply$\&P$) $(K_1$;clo K_2 (lam F_2);$W)$. ΠD_3 : feval $(K_2;W)$ F_2 W_1. ΠD_2 : feval K F_1 W. ΠD_1 : feval K F (clo K_2 (lam F_2)). append C' (c_apply˜ C_3) C $\to$ append (c_app˜ C_1) C_2 C' $\to$ subcomp D_3 C_3 $\to$ subcomp D_2 C_2 $\to$ subcomp D_1 C_1 $\to$ subcomp (evapp D_3 D_2 D_1) C

The result of the transformation into Elf is somewhat shorter:

Elf

```
sc_1   : subcomp (fev_1) (c_1 ~ id).
sc_^   : subcomp (fev_^ D1) (c_^ ~ C1)
            <- subcomp D1 C1.
sc_lam : subcomp (fev_lam) (c_lam ~ id).
sc_app : subcomp (fev_app D3 D2 D1) C
            <- subcomp D1 C1
            <- subcomp D2 C2
            <- subcomp D3 C3
            <- append (c_app ~ C1) C2 C'
            <- append C' (c_apply ~ C3) C.
```

The implementation of the equivalence theorem follows now trivially:

	LF	
cev_complete	:	$\Pi K : \text{env}.\ \Pi F : \text{exp}.\ \Pi W : \text{val}.$ feval $K\ F\ W$ $\to$ ceval $K\ F\ W$ $\to$ type
cevc	:	$\Pi K : \text{env}.\ \Pi F : \text{exp}.\ \Pi W : \text{val}.\ \Pi D : \text{feval}\ K\ F\ W.$ $\Pi C : \text{st (emptys;;}K\text{) (ceval } F\text{\&done) empty} \overset{*}{\Rightarrow} \text{st emptys done (empty;}W\text{)}.$ subcomp $D\ C$ $\to$ cev_complete D (run C)

and finally its implementation in Elf has the form:

Elf

```
cev_complete : feval K F W -> ceval K F W -> type.
cevc         : cev_complete D (run C) <- subcomp D C.
```

Execution in Elf

Elf is a logic programming language. Signatures can be executed. We show the execution of the proof of the equivalence theorem. Assume we are given three λ-expressions:

$$\hat{M}_1 = (\lambda x.\ \lambda y.\ (x\ y))$$
$$\hat{M}_2 = (\lambda x.\ \lambda y.\ x)$$
$$\hat{M}_3 = (\lambda x.\ x)$$

Using adequacy we can represent the three λ-expressions in LF:

$$\begin{aligned} M_1 &= (\text{lam (lam (app } (1\uparrow)\ 1))) \\ M_2 &= (\text{lam (lam } (1\uparrow))) \\ M_3 &= (\text{lam } 1) \end{aligned}$$

Next we determine the meaning of $\hat{M} = ((\hat{M}_1\ \hat{M}_2)\ \hat{M}_3)$ according to the natural semantics. This is done by asking the following query:

$$D : (\text{feval empty } ((M_1\ M_2)\ M_3)\ X)$$

Note, that we use M_1, M_2, M_3 only as abbreviations for LF objects: This query contains only two free logical variables : D and X. Elf answers to this query with the following result:

```
D : feval empty (app (app (lam (lam (app (1 ^) 1))) (lam (lam (1 ^)))) (lam 1))  X.
```

Elf answers this query with the following output:

```
X = clo (empty ; clo empty (lam 1)) (lam (1 ^)),
D =
   fev_app (fev_app fev_lam fev_1 (fev_^ fev_1)) fev_lam
      (fev_app fev_lam fev_lam fev_lam).
yes
```

D corresponds to the proof, that the natural meaning of $((M_1\ M_2)\ M_3)$ is $\{(\cdot; \{\cdot, \Lambda\, 1\}), \Lambda\, 1\!\uparrow\}$. The equivalence theorem says that the operational meaning must be the same. Since we implemented one direction of the equivalence proof, we can transform D into an object, which corresponds to the proof that this value X is the operational meaning of D. Using the `sigma` type — available on the Elf toplevel — we can easily formulate the query as

```
sigma [D: feval empty (app (app (lam (lam (app (1 ^) 1)))
        (lam (lam (1 ^)))) (lam 1))  X] cev_complete D E.
```

The execution of this program yields:

```
E =
   run (c_app ~ c_app ~ c_lam ~ c_lam ~ c_apply ~ c_lam ~ c_lam ~ c_apply
          ~ c_app ~ c_^ ~ c_1 ~ c_1 ~ c_apply ~ c_lam ~ id),
X = clo (empty ; clo empty (lam 1)) (lam (1 ^)).
yes
```

The variable E represents the sequence of CLS-commands to calculate the result. The variable X represents the natural meaning of $\hat{M}$. To check, if this is also the operational meaning, we can derive it by asking the following query: Note that Y is the only logical variable.

```
run (c_app ~ c_app ~ c_lam ~ c_lam ~ c_apply ~ c_lam ~ c_lam ~ c_apply
          ~ c_app ~ c_^ ~ c_1 ~ c_1 ~ c_apply ~ c_lam ~ id)
  : ceval empty (app (app (lam (lam (app (1 ^) 1))) (lam (lam (1 ^)))) (lam 1))  Y.
```

Elf verifies our expectation:

```
Y = clo (empty ; clo empty (lam 1)) (lam (1 ^)).
yes
```

This concludes the section on LF and Elf. For more examples how to use LF type theory and the programming language Elf, we refer the reader to the literature [MP91, Pfe99, Pfe92, Pfe94c, Pfe94b, Pfe95].

2.3 Calculus of Inductive Constructions and Coq

In this chapter we want to present the representation of the language $\mathcal{T}$ in a different type theory: the calculus of inductive constructions (CIC). This signature is then represented in Coq.

This section is divided into two subsections. In the first subsection we introduce the theoretical foundation of the calculus of inductive construction. Originally, Coq was based on the "regular" calculus of construction [CH88, C$^+$95, PM93] but the demand for the notion of inductive types and recursion led to an extension of the calculus and to a new version of Coq: V5.10.

In the second subsection we represent the implement the language $\mathcal{T}$ in Coq. [C$^+$95, LPM94]. Since LF is in some sense a subset of CIC, we are not going into a detailed representation of the different notions in CIC, but we reuse the results from the last section: We only show the representation of the example in Coq and how to use Coq as an assistant. We then will state the equivalence theorem in Coq and prove it using the inference component of Coq.

2.3.1 Calculus of Inductive Constructions

We give a brief overview about the theoretical foundations of the calculus of inductive constructions (CIC). This summary is based on the work of Christine Paulin-Mohring about inductive definitions in the system Coq [PM93] and the Coq user manual [C$^+$95]. Some more work has been done in the area: [Hue88, DH94]. This subsection is divided into two paragraphs. The first paragraph treats the notion of terms, context, and environment. There are some differences in naming between LF and CIC. An *environment* in CIC corresponds roughly to a signature in LF. The second paragraph presents the inference rules for Coq.

Terms

The basic language of CIC is the language of terms. Object, type and kind level can be recovered from the notion of terms by defining external judgments. Terms are defined as follows:

Definition 2.8 (Terms) *A term t in CIC can be formed as*

$$t ::= c \mid s \mid x \mid (x : t_1)t_2 \mid [x : t_1]t_2 \mid (t_1\ t_2)$$

where c is a constant defined in the environment — which be introduced later —, s is a sort. $(x : t_1)t_2$ corresponds to a Π-abstraction. $[x : t_1]t_2$ corresponds to a λ-abstraction. $(t_1\ t_2)$ stands for application.

In LF λ-expressions are defined on the object level. The variable bound by a λ-expression has to be of a type A. In CIC, the λ-expression $[x : t_1]t_2$ expects x to be of type t. Note, that $[x : t_1]t_2$ is also a term t. In LF, there are two different notions of Π-abstraction: Π-abstraction on the type level and Π-abstraction on the kind level. In CIC there is only one Π-abstraction, written as $(x : t)t$. Because of the recursive definition, there are many different layers of types. What is worse: self-referential constructions are possible. To avoid self-referential types, the notion of sorts is introduced. Every term must be of a sort. Since sorts are terms, a well-ordering of sorts is required: This is done by introducing the notion of type universes. Type universes are indexed by natural numbers: Two indexed sorts are defined in CIC: `Type`(i) and `Typeset`(i). There are two basic sorts: `Set` and `Prop`.

The next constructs to be defined are contexts and environments. Contexts are defined similar to LF: A context is a list of variable names and their types. Since there are no types, variables have to be typed with terms.

Definition 2.9 (Context) Γ *is a context :iff*

$$\Gamma ::= \cdot \mid \Gamma, x : t$$

The environment represents all defined constants. Constants can be *declared* to be of type t: $c : t$. In LF two different kinds of constants were introduced by signatures. Type constants and object constants. Since there is no distinction between types and objects from a syntactical point of view, there is only one declaration of constants in CIC. Two non-standard constructions can be found in an environment: the declaration $\texttt{Def}(\Gamma)(c := t_1 : t_2)$ serves to introduce new constants as names for already existing terms. We say constants are *defined.* The declaration $\texttt{Ind}(\Gamma)[\Gamma_P](\Gamma_I : \Gamma_C)$ serves to introduce inductive and mutually inductive types. We say, that by this declaration constants are *inductively defined.* In this definition the first parameter Γ represents the context, in which the inductive type is to be defined. Γ_P stands for a set of parameters. This allows the definition of generic types. Γ_I stands for a context of definitions. In a simple setting one would expect Γ_I to contain only one element, namely the constant to be defined. The problem arises with mutually recursive types. By making Γ_I a context simultaneous definitions of mutually inductive types are possible. The context Γ_C represents a set of constructors.

Definition 2.10 (Environment) E *is an environment :iff*

$$E ::= \cdot \mid E, c : t \mid E, \texttt{Def}(\Gamma)(c := t : t) \mid \texttt{Ind}(\Gamma)[\Gamma](\Gamma : \Gamma)$$

These are the basic notions we need to define the judgements and inference rules in the next paragraph.

Rules

In this section we will introduce briefly the judgements concerning typing. We follow the presentation in [C+95], chapter 6.

The first judgement is concerned with the well-formedness of an environment E and a context Γ which possibly depends on E. This judgement has the form

$$\mathcal{WF}(E)[\Gamma]$$

It corresponds to the LF judgement $\vdash \Gamma$ ctx.

The second judgement stands for well-typedness. It expresses that a term t is of type T with t, T terms built from constants and variables defined in E and Γ. The judgment has the general form:

$$E[\Gamma] \vdash t : T$$

It would be beyond the aim of this chapter to give a complete overview over all typing rules. We will restrict ourselves to few of them. The aim is to sketch the idea. For a more complete presentation consult [C+95].

Well-formedness rules First we present the base case. The rule following says that the empty context and the empty environment are always well-formed.

$$\frac{}{\mathcal{WF}([])[[]]}\,\textsf{wfemp}$$

From the definition of environment follows, that there are three different ways to declare constants: Declaration, definition and inductive definition. We examine well-formedness for definitions and inductive definitions:

A constant definition is of the form $\mathtt{Def}(\Gamma)(c := t : T)$. t, T are both terms, t stands for an object of type T. Since t is a term, it can take different forms. There is not one well-formed rule for definitions but one for every form of t. We present the rule in the case of t being an abstraction.

$$\frac{\mathcal{WF}(E; \mathtt{Def}(\Gamma; x : U)(c := t : T); E')[\Delta] \quad \mathcal{WF}(E)[\Gamma]}{\mathcal{WF}(E; \mathtt{Def}(\Gamma)(c := [x : U]t : (x : U)T); [c/(c\,x)](E'))[[c/(c\,x)](\Delta)]}\,\textsf{wfdeflam}$$

This rule reads as follows: If E and Γ are well-formed, and the extension of environment E by a new constant definition $c := t : T$ in a new context Δ is well-formed, then the environment E extended by the definition of c being $[x : U]t$ of type $(x : U)T$ is well-formed in Δ. $[x : U]t$ corresponds to a λ-abstraction, $(x : U)T$ corresponds to the formation of a Π-type. Since c can occur free in the remaining environment E' and the context Δ, all occurrences of c have to be replaced by c applied to x in E' and Δ. We remark, that t and T can contain free variables from context Γ.

Next we address the problem of well-formedness of inductive definitions. Before we describe the rule, we have to define some more concepts:

We have seen in LF that types and dependent types are also "typed". These types are called kinds. Since the distinction vanishes in Coq an auxiliary notion has to be introduced, the notion of sort and *arity.* A term t is and arity of sort s, when it is either a sort itself or a Π-closure of this sort. Arities are therefore defined as:

Definition 2.11 (Arity) *T is an arity of sort s :iff*

$$T = s \text{ or } T = (x : U)T' \text{ with } T' \text{ arity of sort } s$$

In LF there is the notion of canonical forms. Without going into details, the canonical form of a type is always something like

$$(x_1 : T_1)..(x_k : T_k\,)I\; t_1...t_n$$

This term is a *type of constructor* of I, because when instantiated appropriately, it generates an instance of type $I\ t_1 ... t_n$. We define the type of constructor more formally after the following example:

Example 2.12 *Consider the following example: Define* exp *as a type constant of sort* type(1)*. Then the following term is an arity of sort* type(1) *in CIC:* $(e : \text{exp})(v : \text{exp})\ \text{type}(1)$*; we can define a new type constant:*

$$\text{eval} : (e : \text{exp})(v : \text{exp})\ \text{type}(1)$$

eval *is a dependent type. Object constants are also referred to as constructors in CIC. In the example* $\text{eval\_s} : (e : \text{exp})(v : \text{exp})(d : \text{eval}\ e\ v)\ \text{eval}\ (\text{s}\ e)\ (\text{s}\ v)$ *defines the constructor* eval_s. $(e : \text{exp})(v : \text{exp})(d : \text{eval}\ e\ v)\ \text{eval}\ (\text{s}\ e)\ (\text{s}\ v)$ *is a* type of constructor of eval.

Here the formal definition:

Definition 2.13 (Type of constructor) *T is a type of constructor of I :iff*

$$T = (I\ t_1\ t_2\ ...\ t_n) \textit{ or } T = (x : U)T' \textit{ with } T' \textit{ type of constructor of } I$$

Finally we have to define the so-called *positivity condition.* This condition says that

Definition 2.14 (Positivity condition) *T satisfies the positivity condition with respect to a constant X :iff*

1. *if $T = (T'\ t_1 ... t_n)$ then X does not occur in $t_1 ... t_n$*
2. *if $T = (x : U)T'$ then U, T' satisfy the positivity condition with respect to X*

There is also a strict positivity condition:

Definition 2.15 (Strict positivity condition) *T satisfies the strict positivity condition with respect to a constant X :iff*

1. *if $T = (T'\ t_1 ... t_n)$ then X does not occur in $t_1 ... t_n$*
2. *if $T = (x : U)T'$ then X does not occur in U and T' satisfy the positivity condition with respect to X*

With these definition we can now address the well-formedness of inductive definitions. Let $\Gamma, \Gamma_P, \Gamma_I, \Gamma_C$ be contexts. Γ, Γ_P are contexts in which the definition takes place, we do not need to examine their structure. Γ_I stands for the set of defined inductive constants and their types: $\Gamma_I := a_1 : A_1; ...; a_k : A_k$. Γ_C is the context which defines the constructors for the inductively types: $\Gamma_C := c_1 : C_1; ... c_n : C_n$. E is the environment.

$$\frac{\mathcal{WF}(E)[\Gamma] \quad (E[\Gamma; \Gamma_P] \vdash A_j : s'_j)_{j=1..k} \quad (E[\Gamma; \Gamma_P; \Gamma_I] \vdash C_i : s_i)_{i=1..n}}{\mathcal{WF}(E; \texttt{Ind}(\Gamma)[\Gamma_P](\Gamma_I : \Gamma_C))[\Gamma]}\ \textsf{wfind}$$

provided that

- s'_j, s_i are sorts
- a_j, c_i are different names ($j = 1..k$, $i = 1..n$)
- A_j is an arity of type s'_j and $a_j \notin \Gamma \cup \Gamma_P \cup E$, ($j = 1..k$)
- C_i is a type of constructor of a_j (for some $j \leq k$), which satisfies the positivity condition for $a_1..a_k$ and $c_i \notin \Gamma \cup \Gamma_P \cup E$ ($i = 1..n$)

This rule reads as follows: To prove the well-formedness of an inductive definition we have prove that first the environment and the context Γ are well-formed, that means that the setting in which the definition takes place is represented in CIC. Second, it has to be checked if the types newly introduced by the definition are actually types in the current setting. To do so, it must be checked that every A_j is of sort s'_j, in the context $\Gamma; \Gamma_P$. Finally it has to be checked that the constructor types are types. Note that since mutual dependencies are possible, the context is extended by Γ_I. We obtain $\Gamma; \Gamma_P; \Gamma_I$ as actual context. The side conditions ensure, that every declaration in Γ_C contributes to the definition of a type in Γ_I.

Rules for well typedness: As mentioned above, besides the well-formedness judgment, there is also a well-typedness judgment: $E[\Gamma] \vdash t : T$. This judgement corresponds to the LF judgment $\Gamma \vdash_\Sigma M : A$. It expresses the property, that a term t has type T in an environment E, and context Γ. In the following, we will present a few selected inference rules, which should serve for two purposes. First we want to show the relationship between well-typedness and well-formedness and second we want to present the most important features of the semantics.

The following two rules show that if an environment and a context are well-formed, then it is possible to extract typing information from either of them:

$$\frac{\mathcal{WF}(E)[\Gamma] \quad (x : T) \in \Gamma}{E[\Gamma] \vdash x : T}\,\textsf{tpvar} \qquad \frac{\mathcal{WF}(E)[\Gamma] \quad (c : T) \in E}{E[\Gamma] \vdash c : T}\,\textsf{tpconst}$$

In the case of abstraction we obtain the following rule:

$$\frac{E[\Gamma] \vdash (x : T)U : s \quad E[\Gamma; x : T] \vdash t : U}{E[\Gamma] \vdash [x : T]t : (x : T)U}\,\textsf{tplam}$$

The rule reads as follows: If a λ-expression $[x : T]t$ is to be checked to be of Π-type $(x : T)U$ then two conditions have to be verified: First, $(x : T)U$ must be a type, i.e. it must be proven to be of a sort s. This must be done, because there are types which look correct, but are not because of self-reference. Second, t must be of type U, assuming x to be of type T in context Γ.

If the rules are read from top to bottom, we see that the rules represent a logic: The type $(x : T)U$ can be read as a universal quantified formula. When $(x : T)U$ is a well-formed formula, and a proof term for $(x : T)U$ can be derived, then we consider $(x : T)U$ to be true. The object $[x : T]t$ can therefore be interpreted as a program or as a proof term of the statement *forall x of type T, U holds.*

We address the problem of typing with inductive definitions: Assume that we are working in a context Γ with r parameters — defined in the context Γ_P. The context of inductively defined constants is defined as $\Gamma_I = a_1 : A_1 ... a_k : A_k$. For every inductively defined constant a_k we define a set of constructor terms. All these constructor terms and their types are summarized in the context $\Gamma_C = c_1 : C_1; ... c_n : C_n$. Let $\texttt{Ind}(\Delta)[\Gamma_P](\Gamma_I := \Gamma_C)$ an inductive definition in the environment. Every type C_i is a type of constructor of an a_j. The form of C_k is implicitly given as

$$(x_1 : T_1^p)..(x_r : T_r^p)(y_1 : T_1^{(i)})..(y_{m_i} : T_{m_i}^{(i)})\ a_j\ t_1^{(i)}..t_{l_j}^{(i)}$$

Variables introduced in the Π-closure of C_k may depend on types which are defined in Γ_I. These variables are called *recursive.*

To perform a proof over a mutual inductively defined type, k properties $P_1..P_k$ have to be proven. The property P_i expresses something about the inductive type a_i. It depends on all arguments of type a_i — that is on l_i terms $t_1..t_{l_i}$ — and on the object the property P_i should be proven for. The destructor proof term of an inductive type has the form:

$$\langle P_1 ... P_r \rangle\ \texttt{Match}\ c\ \texttt{with}\ f_1 ... f_n\ \texttt{end}$$

c is assumed to be constructed by one of the constructors $c_i : C_i$ in Γ_C. Therefore, it must be of the form

$$c_i\ q_1..q_r\ a_1..a_{m_i}$$

The f_i represent proof objects of different cases of a derived induction principle. We will see shortly, how induction principles are generated. The operational meaning of this proof term is, that by means of the form of the constructor c_i the proof term f_i can be selected, and the Match-expression can be reduced. This is called as ι-reduction. A more detailed presentation of reduction is given in [C+95].

Now we address the definition of the typing rule for inductive definitions by itself. Assume we have a derivation of a term of a type defined within the inductive type:

$$E[\Gamma] \vdash c : (a_u\ q_1 ... q_r\ t_1 ... t_s)$$

First we have to show, that the properties P_j's are well-formed types, that is, they must be of sort B_j, for $j \leq k$. We also have to show that the the sort of $(a_i\ q_1 ... q_r)$ is "bigger" than the sort B_i. We omit the details, the reader may consult [C+95]. Next we have to find proof terms f_i for every induction principle derived by the constructors c_i: The induction principles are derived by a rather technical construction of $\{c : C\}_{a_1 ... a_k}^{P_1 ... P_k}$:

Definition 2.16 (Induction principle) *The induction principle is defined by*

$$\begin{array}{rcll} \{c : (a_i\ q_1 ... q_r\ t_1 ... t_{l_i})\}_{a_1 ... a_k}^{P_1 ... P_k} & \equiv & (P_i\ t_1 ... t_{l_i}\ c) & \\ \{c : (x : T)C\}_{a_1 ... a_k}^{P_1 ... P_k} & \equiv & (x : T)\{(c\ x) : C\}_{a_1 ... a_k}^{P_1 ... P_k} & x \textit{ non-recursive} \\ \{c : (x : T)C\}_{a_1 ... a_k}^{P_1 ... P_k} & \equiv & (x : T)\{x : T\}_{a_1 ... a_k}^{P_1 ... P_k} \rightarrow \{(c\ x) : C\}_{a_1 ... a_k}^{P_1 ... P_k} & x \textit{ recursive} \end{array}$$

The first case says that an object of the type A_i is replaced by the proposition P_i. The second case expresses that Π-abstraction must be interpreted as universal quantification under the assumption that x is not recursive. If it is recursive, i.e. its type is defined by means of Γ_I, the induction hypothesis has to be provided: this is expressed by the third case. We now sketch the typing rule for inductive definitions:

$$\frac{E[\Gamma] \vdash c : (a_u\ q_1..q_r\ t_1..t_2) \quad (E[\Gamma] \vdash P_j : B_j)_{j \leq k} \quad (E[\Gamma] \vdash f_i : \{(c_i\ q_1..q_r) : (C_i\ q_1..q_r)\}^{P_1..P_k}_{a_1..a_k})_{i \leq n}}{E[\Gamma] \vdash \langle P_1..P_k \rangle\ \texttt{Match}\ c\ \texttt{with}\ f_1..f_n\ \texttt{end} : (P_u\ t_1..t_{l_u}\ c)}\ \textsf{tpind}$$

The following example shows a simplified version of **tpind**:

Example 2.17 *Consider the simple inductive definition of the natural numbers: as before we define a type* exp : type(1)*. The inductive type of natural numbers has the following form:*

$$\mathcal{I} := \texttt{Ind}(\emptyset)[\emptyset]((\mathrm{exp} : \mathrm{type}(0)) := (\mathrm{z} : \mathrm{exp}; \mathrm{s} : \mathrm{exp} \to \mathrm{exp}))$$

The environment consists of exp : type(1) *and* $\mathcal{I}$*. The variable* P *stands for the property we try to prove. It expects only one argument: a natural number:* $P : \mathrm{exp} \to s$*, where* s *is a sort. The construction of the induction principles yields the following result:*

$$\begin{aligned}
\{\mathrm{z} : \mathrm{exp}\}^P_{\mathrm{exp}} &= (P\,\mathrm{z}) \\
\{\mathrm{s} : \mathrm{exp} \to \mathrm{exp}\}^P_{\mathrm{exp}} &= \{\mathrm{s} : (e : \mathrm{exp})\ \mathrm{exp}\}^P_{\mathrm{exp}} \\
&= (e : \mathrm{exp})\{e : \mathrm{exp}\}^P_{\mathrm{exp}} \to \{(\mathrm{s}\ e) : \mathrm{exp}\}^P_{\mathrm{exp}} \\
&= (e : \mathrm{exp})(P\ e) \to (P\ (\mathrm{s}\ e))
\end{aligned}$$

Finally we describe a simplified version of the typing rule for induction: **tpind***. Note, that* E *contains only the inductive definition* $\mathcal{I}$ *and* Γ *is empty. We omit therefore* $E[\Gamma]$*. The version of the match rule for this example has the simplified form:*

$$\frac{\vdash c : exp \quad \vdash P : exp \to s \quad \vdash f_1 : (P\ z) \quad \vdash f_2 : (e : exp)(P\ e) \to (P\ (s\ e))}{\vdash \langle P \rangle\ \texttt{Match}\ c\ \texttt{with}\ f_1, f_2\ \texttt{end} : (P\ c)}\ \textsf{tpind}$$

This concludes our presentation of the calculus of inductive constructions. In the next subsection we describe Coq.

2.3.2 Coq

In this section we will present an implementation of the language $\mathcal{T}$ from section 2.1 using Coq. As in section 2.2.2, we implement de Bruijn expressions, the notion of natural and operational semantics and the equivalence theorem. We will use Coq's inference engine to prove the append lemma and the equivalence theorem.

How to use Coq

Coq V5.10 is a proof assistant. It is a direct implementation of CIC. Coq has a very sophisticated interface to the user. Because an inference component is included in the distribution, the command language of Coq is equipped with many features, which are not used in this presentation. There are at least two different ways to define inductive types: Inductively and recursively. We omit all details of how to use Coq, and refer the interested reader to [C+95]. In the remainder of this subsection we show how to use features of Coq when they are needed.

Implementation of $\mathcal{T}$

We implement de Bruijn expressions by the inductive type exp. It must be defined inductively, because app depends on exp: Inductive definitions are easily represented in Coq. Here is the implementation:

```
Inductive
  exp: Set :=
      top : exp
    | pop : exp -> exp
    | app : exp -> exp -> exp
    | lam : exp -> exp.
```

Environment and values have to be implemented as inductive types, too. Recall, that the notion of environment cannot be defined as a stack of expressions. An environment is a stack of values, values are closures of environments and expressions. Both notions have to be implemented by mutual induction. CIC and Coq support mutually dependent inductive definitions. Values are implement as type val. Environments are implemented as type env:

```
Mutual Inductive env: Set :=
      empty : env
    | cons  : env -> val -> env
with
  val: Set :=
      clo   : env -> exp -> val.
```

Implementation of the Natural Semantics

In this paragraph we show the implementation of the evaluation judgement eval : env $\rightarrow$ exp $\rightarrow$ val $\rightarrow$ type. Recall, that the first parameter stand for the actual environment[1]. The second parameter represents the program to be executed. The third argument represents the natural meaning of the second.

The evaluation judgment is implemented as the inductive type eval. Note that in terms of CIC eval is a type constant of the term env -> exp -> val -> Prop which is an arity of sort Prop.

```
Inductive
  eval   : env  -> exp -> val -> Prop :=
      ev_top : (K: env)(W: val)(eval (cons K W) top W)
    | ev_pop : (K: env)(F: exp)(W: val)(W': val)
                 (eval K F W) -> (eval (cons K  W') (pop F) W)
    | ev_lam : (K: env)(F: exp)(eval K (lam F) (clo K (lam F)))
    | ev_app : (K: env)(K':env)(F1: exp)(F1': exp)(F2: exp)(W: val)(W2: val)
               (eval K F1 (clo K' (lam F1')))
```

[1]in the sense of section 2.1

```
                -> (eval K F2 W2)
                -> (eval (cons K' W2) F1' W)
                -> (eval K (app F1 F2) W).
```

All four cases correspond directly to the LF types, we introduced in section 2.2.2. It is noteworthy, that we do not use the additional expressive power of CIC. We represent LF types in CIC by implementing them in Coq.

Unfortunately, Coq does not have an appropriate type reconstruction algorithm which would allow to omit the Π-closure around the newly defined constructor types. It is easy to see, that the Coq implementation of our example lacks some elegance compared to the representation in Elf.

Coq offers a certain kind of remedy for this problem: syntactic definitions. Syntactic definitions help to hide unnecessary arguments of constants. Unnecessary in a way that Coq can derive the parameters which have been omitted by type inference. But one cannot omit Π closures, when implementing types in Coq. We implement the following syntactic definitions.

```
Syntactic Definition i_ev_top := (ev_top ? ?).
Syntactic Definition i_ev_pop := (ev_pop ? ? ? ?).
Syntactic Definition i_ev_lam := (ev_lam ? ?).
Syntactic Definition i_ev_app := (ev_app ? ? ? ? ? ? ?).
```

Implementation of the CLS Machine

In the first two paragraphs we described an implementation of $\mathcal{T}$ and an implementation of the natural semantics of $\mathcal{T}$. We will now focus on the operational aspects of the language. Here is the implementation of the CLS machine.

We need the notion of environment stacks. Environment stacks are represented as an inductive type envstack.

```
Inductive
  envstack : Set :=
      emptys : envstack
    | conss : envstack -> env -> envstack.
```

We have seen in section 2.1, that there are two different versions of instructions. De Bruijn expressions are instructions and special keywords which can combine subcomputations are instructions.

The way how we represent instructions is as follows. We define a inductive type instruction. The first kind of instructions is defined using the embedding function ev. The instruction for combining the two subcomputation for application is represented by the constant apply.

```
Inductive instruction : Set :=
  apply : instruction
| ev : exp -> instruction.
```

A program is now viewed as a list of instructions. done signals the end of a computation.

```
Inductive
  program: Set :=
    done   : program
  | consp : instruction -> program -> program.
```

A state consists of a stack of environments which stores backup copies of the actual environment, the program as an instruction list and a result stack — in form of an environment: It represents intermediate subcomputation results. States are implemented as type `state`.

```
Inductive state: Set :=
  st : envstack -> program -> env -> state.
```

Implementation of $\mathcal{T}$'s Operational Semantics

In this paragraph we implement the notion of computation. For this purpose we defined two type-families in section 2.2.2, which represent single step transitions, and multi-step transitions. Both notions lead quite naturally to the definition of program evaluation with respect to a CLS machine, that is the operational semantics.

The type representing single step transitions need not to be defined inductively. For the sake of continuity, we implement it as the inductive type `single`:

```
Inductive single : state -> state -> Prop :=
  c_top : (Ks: envstack)(K:env)(W:val)(P:program)(S:env)
          (single (st (conss Ks (cons K W)) (consp (ev top) P) S)
            (st Ks P (cons S W)))
| c_pop : (Ks: envstack)(K:env)(W':val)(F:exp)(P:program)(S:env)
          (single (st (conss Ks (cons K W')) (consp (ev (pop F)) P) S)
            (st (conss Ks  K) (consp (ev F) P) S))
| c_lam : (Ks: envstack)(K:env)(F:exp)(P:program)(S:env)
          (single (st (conss Ks K) (consp (ev (lam F)) P) S)
            (st Ks P (cons S (clo K (lam F)))))
| c_app : (Ks: envstack)(K:env)(F1:exp)(F2:exp)(P:program)(S:env)
          (single (st (conss Ks K) (consp (ev (app F1 F2)) P) S)
            (st (conss (conss Ks  K) K) (consp (ev F1)
              (consp (ev F2) (consp apply P))) S))
| c_apply : (Ks: envstack)(K':env)(F1':exp)(W2:val)(P:program)(S:env)
            (single (st Ks (consp apply  P)
              (cons (cons S (clo K' (lam F1'))) W2))
              (st (conss Ks  (cons K' W2)) (consp (ev F1') P) S)).
```

For better readability we introduce the following syntactic definitions for the single step constructors.

```
Syntactic Definition i_c_top   := (c_top ? ? ? ? ?).
Syntactic Definition i_c_pop   := (c_pop ? ? ? ? ? ?).
Syntactic Definition i_c_lam   := (c_lam ? ? ? ? ?).
Syntactic Definition i_c_app   := (c_app ? ? ? ? ? ?).
Syntactic Definition i_c_apply := (c_apply ? ? ? ? ? ?).
```

The multi step transition relation is only the transitive closure of the single step transition relation: It is implemented in Coq as an inductive type multi. It must be represented as an inductive type because consm depends on multi.

```
Inductive multi : state -> state -> Prop :=
    id : (St:state)(multi St St)
  | consm : (St:state)(St':state)(St'':state)(single St St')
            -> (multi St' St'')
            -> (multi St St'').
```

This definition requires again some syntactic definitions:

```
Syntactic Definition i_id    := (id ?).
Syntactic Definition i_consm := (consm ? ? ?).
```

Finally we can define the operational meaning of a de Bruijn expression: we implement the inductive type ceval, which depends on the actual environment, the expression which is to be evaluated. The result is a value. Note, it is not necessary to define ceval, inductively.

```
Inductive ceval : env -> exp -> val -> Prop :=
  run : (K:env)(F:exp)(W:val)
          (multi (st (conss emptys  K) (consp (ev F) done) (empty))
             (st (emptys) (done) (cons empty  W)))
     -> (ceval K F W).
```

Here again, we need to introduce a syntactic definition.

```
Syntactic Definition i_run := (run ? ? ?).
```

Implementation of the Equivalence Theorem

In this paragraph we derive the one direction of the equivalence proof with support of the inference engine of Coq. We first state and prove the append lemma, which is needed in the proof of the equivalence theorem.

The definition of the type multi is based on the idea, that a trace is defined if and only it is either empty or the first step of the trace is a single step transition and the rest is a trace. The append lemma guarantees that two traces can be concatenated.

The formulation of the lemma is as follows.

```
Lemma append:  (A:state)(B:state)(C:state)
  (multi A B) -> (multi B C) -> (multi A C).
```

The proof of the theorem is straightforward. The only noteworthy step is the application of the consm constructor. Here the intermediate state has to be provided by the user. The proof in Coq has the following form.

```
Intros A B C H.
  Elim H.
    Auto.
  Intros. Apply consm with St'. Auto.
    Auto.
Qed.
```

The command Intros applies four times the Π-introduction rule. Elim applies a destructor rule to the inductively defined (multi A B). For a more detailed description of the commands consult [C+95].

The equivalence theorem states the following fact: When a de Bruijn expression F evaluates to a value W, in context K, then there is a computation trace of a CLS machine, i.e. $K \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\!\rightarrow} W$. We need a stronger induction hypothesis, so we proved the subcomputation lemma 2.1 in section 2.1. Here is the formulation of this lemma:

```
Lemma subcomp:
  (K:env) (F:exp) (W:val)
  (eval K F W) -> (Ks:envstack) (P:program) (S:env)
                        (multi (st (conss Ks  K) (consp (ev F)  P) S)
                        (st Ks P (cons S W))).
```

To make the proof easier, we provide the constants to the system, which should be automatically applicable. We do this by using the hint command of Coq.

```
Hint c_top c_pop c_lam c_app c_apply id.
```

Here is the proof. The proof again is straightforward. The problem of finding the proof *fast* lies in the hints the user has to give to the system. As the proof shows, 9 different states have to be calculated by hand and provided to system. This makes the proof quite complex.

```
Intros K F W H. Elim H.
  Intros. Apply consm with (st Ks P (cons S W0)). Auto.
    Auto.
  Intros. Apply consm with (st (conss Ks K0) (consp (ev F0) P) S). Auto.
    Auto.
  Intros. Apply consm with (st Ks P (cons S (clo K0 (lam F0)))). Auto.
    Auto.
  Intros.
    Apply consm with (st (conss (conss Ks K0) K0)
                   (consp (ev F1) (consp (ev F2) (consp apply P))) S) . Auto.
    Apply append with (st (conss Ks K0) (consp (ev F2) (consp apply P))
                  (cons S (clo K' (lam F1')))). Auto.
    Apply append with (st Ks (consp apply P)
                  (cons (cons S (clo K' (lam F1'))) W2)). Auto.
    Apply consm with (st (conss Ks (cons K' W2)) (consp (ev F1') P) S). Auto.
    Apply append with (st Ks P (cons S W0)). Auto.
    Auto.
Qed.
```

In section 2.1 we saw, that the proof of one direction of the equivalence theorem is a direct consequence of the subcomputation lemma:

```
Theorem completness :
  (K:env) (F:exp) (W:val)
  (eval K F W) -> (ceval K F W).

Hint run subcomp.
Qed.
```

This concludes the presentation of the $\mathcal{T}$ in Coq. In the next chapter we will define the meta logical framework MLF, which can be used as the theoretical foundation of a proof development environment based on LF.

2.4 Result

In this chapter we showed, that Elf is a logic programming language based on LF. It does not offer any mechanism to use it as an automated theorem proving system. On the other hand we showed, that Coq performs very good in automated theorem proving issues, but it is too powerful to be used as a programming language. The aim of this thesis is to propose a system, which equips Elf with an appropriate meta logic. This meta logic should be so powerful, that proofs by induction can be handled easily. It should not be too powerful to prevent inconsistencies.

We remarked that LF is not as powerful as CIC. For a more detailed investigation see [Bar92]. The representation of the example from section 2.1 in Coq followed closely the representation in LF.

In the next chapter we will present such a meta logic for the Horn fragment of LF: It is called MLF. MLF can be seen as a sequent calculus on top of LF. It supports reasoning over LF signatures. Induction as a fundamental proof technique is represented by a special rule, the case rule. In contrast to the approach realized in CIC, we omit the generation of induction principles. Instead we allow recursion which usage is restricted to avoid the generation of non total proof objects. We believe that the explicit generation of induction principles limits the power of the inductive component, and we also believe that by omitting these principles the expressive power of the meta logic is increased.

A interactive proof system for MLF on top of LF is not yet implemented. We show in chapter 5 how MLF can be used to prove the meta theoretical results.

Chapter 3

MLF

In this chapter we introduce MLF. MLF is a meta logic which allows reasoning about the Horn fragment of the logical framework LF. The proof theory of MLF is given as an intuitionistic sequent calculus [Gal93], equipped with rules to reason about LF types and LF objects. Since LF is very frequently used to represent deductive systems, induction is a major concept in MLF. The notion of induction differs from others [C^+95]. A standard approach would be to introduce induction in form of induction principles. An induction principle corresponds to proof by structural induction over the structure of a term. The main disadvantage of the generation of induction principles is the inflexibility which arises because induction hypothesis can only be applied to direct subterms. A lot of proofs can only be done by *complete* structural induction: the induction hypothesis must be applicable to *any* smaller term according to a well-founded ordering. If this kind of induction is used, the proof of being "smaller" has to be performed within the meta logic.

The rules of MLF are equipped with proof terms. This is the motivation to prove the "smaller" relation on the basis of the proof terms — outside of MLF. Hence we define a rule which provides the induction hypothesis with this requirement formulated as a side condition. The second important rule to complete the treatment of induction is the case distinction rule. This rule allows to discriminate between different forms of LF-objects. Similar ideas can be found in [MN94].

This chapter is organized as follows: In the first section we introduce the language of MLF, the Horn fragment of LF and proof terms. We introduce basic notions like substitution and unification. In the second section we introduce the inference rule system for MLF and in the last we demonstrate how to use it.

3.1 Language

In this section we define the language of MLF. The calculus incorporates two levels of reasoning. On the meta level we reason about formulas and proof terms, on the LF level about types and objects of the Horn fragment of LF type theory.

MLF is restricted to the Horn fragment of LF because of these two levels of reasoning: Because of the strict distinction between both levels, we define two totally distinct variable concepts: One variable concept for the meta level and one variable concept for the LF level. As we will see, it is impossible to construct objects for a function type in LF in general. We will discuss this problem in more detail, when we describe the typing rules for MLF.

When we reason in MLF, we will keep track of a set of assumptions and a goal, for which a proof term is to be constructed. Since function types of LF cannot occur as a goal, the notion of types in the Horn fragment of LF is split into two notions: LF types which can occur as assumptions, and LF types which can occur as goals.

The distinction of two different LF types gives reason to discriminate over MLF formulae. We make a difference between formulae which occur as assumptions which we called *data formulae*, and formulae which occur as goals which we call *goal formulae.* We address now the exact characterization of the meta level and the LF level.

Meta level:

The meta level stands for reasoning about LF types and LF objects. The meta level by itself consists of two different layers. There are *formulae*, which represent properties of LF types and LF objects on the meta level. Furthermore, there are proof terms, which correspond to proofs of formulae. A proof term captures the computational content of a proof. It are called *program.*

In the next paragraph we will introduce the LF level. Since LF types and LF objects are the entities which should be reasoned about, there must be an interface between the meta level and the LF level: We define therefore a non-standard kind of formula which represents LF types.

If A is an LF type, then $\overline{A}$ denotes the corresponding formula. The function $\overline{\cdot}$ is an embedding function of LF types into formulae.

A different connection has to be established between LF objects and programs. LF objects are considered as proofs for embedded LF types. If M is an LF object, $\overline{M}$ is the corresponding program.

The motivation for this construction is as follows: In general in logic a formula is provable if a derivation can be found using a complete and sound calculus. In this general case, there are only two possibilities: A formula is provable or not. The demands towards MLF are much more general in this respect. MLF should provide an answer for the question: *Why is a formula provable?* This immediately raises the question what it means for an embedded LF type to be true: An embedded LF type is considered to be true if and only if it is inhabited. The question for the *Why* can be answered by pointing to the *witness* object.

Let M be a proof object which witnesses A to be inhabited. We write $\overline{M}$ as the representation of the LF object as a program: $\overline{M}$ is a proof of $\overline{A}$ if and only if M is an object of type A. A picture can make this more clear:

$$\begin{tikzcd}
\text{Programs} & \text{Formulae} \\
\text{LF objects} \arrow[u, "\overline{\cdot}"] & \text{LF types} \arrow[u, "\overline{\cdot}"]
\end{tikzcd}$$

The variable concept on the meta level and the variable concept on the LF level are two totally different concepts. They should not be mixed up. We denote the set of meta variables with X. To make the distinction between meta variables and object variables clearer, we denote meta variables always with uppercase letters and object variables always with lowercase letters.

We define now the meta logic. As mentioned earlier, the meta logic is based on first order intuitionistic logic, taking into account the Horn fragment of the underlying type theory LF. The language of formulae contains universal and existential quantification, conjunction, disjunction and implication. 1 stands for "True". First we define the most general notion of formula, we call it F:

$$\text{Formulae:} \quad F \quad ::= \quad \forall X : A.F \mid \exists X : A.F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid 1 \mid \overline{A}$$

Then we restrict this concept to the restricted versions briefly introduced above: goal formulae and data formulae: The notion of LF type is divided into types which can occur as goal types A_G and types which can occur as a data types A_D.The goal formulae are defined as follows:

$$\text{Goal Formulae:} \quad G \quad ::= \quad \forall X : A_D.G \mid \exists X : A_G.G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \rightarrow G \mid 1 \mid \overline{A_G}$$

and the data formulae are defined as

$$\text{Data Formulae:} \quad D \quad ::= \quad \forall X : A_G.D \mid \exists X : A_D.D \mid D_1 \wedge D_2 \mid D_1 \vee D_2 \mid G \rightarrow D \mid 1 \mid \overline{A_D}$$

Universal quantification, existential quantification, and implication have to be defined this way: this will become evident, when we present the typing rules for programs. Data formula can only occur as assumptions from which a goal formula is to be proven. Without going into details here, there will be some rules in the inference system of MLF which have to be restricted to certain formulae which exist in the intersection of goal formulae and data formulae: One rule for example will allow the actual goal to be transformed into an assumption — this is necessary to provide induction hypothesis as we will discuss later. Therefore we have to characterize an intersection set of goal and data formulae. Since goal formulae and data formulae may be constructed from goal types and data types, a notion of type must be established which describes the set of LF types, which are simultaneously representable as data types and goal types: This set is called A_P. It turns out, that these are exactly the atomic types. It is now straightforward to define the language C of core formulae, which are simultaneously goal formulae and data formulae:

$$\text{Core Formulae:} \quad C \quad ::= \quad \forall X : A_P.C \mid \exists X : A_P.C \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid C_1 \rightarrow C_2 \mid 1 \mid \overline{A_P}$$

All inference rules in MLF are decorated with proof terms. The inference rules concerning the provability judgment in MLF, which will be introduced in the next section, are decorated with proof terms which we call *programs* — to reflect the computational character:

$$\begin{array}{lll} \text{Programs:} \quad P & ::= & X \mid (\texttt{unit}) \mid (\texttt{rec}\ X.P) \mid (\texttt{fun}\ X.P) \mid (\texttt{pair}\ P_1\ P_2) \mid (\texttt{inl}\ P) \\ & & \mid (\texttt{inr}\ P) \mid (\texttt{inx}\ P_1\ P_2) \mid (\texttt{let}\ P_1\ \texttt{be}\ X\ \texttt{in}\ P_2) \mid (\texttt{app}\ P_1\ P_2) \mid \overline{M} \\ & & \mid \left(\begin{array}{l} \texttt{case}\ P\ \texttt{of} \\ \quad Q^{(1)} \Rightarrow P^{(1)} \\ \quad \vdots \\ \mid \quad Q^{(n)} \Rightarrow P^{(n)} \end{array} \right) \end{array}$$

The `case` construct is defined using patterns Q. Ideally we try to achieve every possible program to serve as a pattern. We assume programs to be closed with respect to LF variables.

Therefore we have to restrict the set of possible patterns, to guarantee this assumption to hold: In the following we define patterns on the program level. The variable N refers to a pattern on the object level which we will define below.

$$\text{Program Patterns:}\quad Q \ ::= \ (\texttt{unit}) \mid (\texttt{pair}\, X_1\, X_2) \mid (\texttt{inl}\, X) \mid (\texttt{inr}\, X) \mid (\texttt{inx}\, X_1\, X_2) \mid \overline{N}$$

LF level:

MLF is designed to reason about LF. Therefore we have to distinguish quite carefully between the LF level and meta level. We address now the definition of the LF level. We distinguish between objects, types and kinds. Note, that the Horn fragment of LF type theory is a straightforward restriction of LF type theory. As introduced in section 2.2.1 we use x to denote LF variables, c to denote object constants and a to denote type constants. The object level of LF has to be extended with a projection function: Programs can be used as objects. Consider the following short example:

Example 3.1 *Let Σ be an LF signature defining two type constants: the constant* exp *which stands for natural numbers and the constant* val *which represents a judgment saying that an expression is a value.* val *is a dependently typed constant. Σ also defines the object constants zero* z *and the successor function* s*. See [Pfe92] for more detail. Assume that there is a meta variable X which represents a proof term for the formula $\overline{\text{exp}}$. This reads as: X stands for a witness, that $\overline{\text{exp}}$ is inhabited, moreover it represents a proof object of the form $\overline{M}$.*

The objective is to express the following statement on the meta level: if X is an expression and X is a value then the successor of X is also a value. Assume we have X, a proof of the formula $\overline{\text{exp}}$. How can we express the second assumption, that X is a value? On the LF level it is quite clear how to do it: If x is a object and y is an object of $\overline{(\text{val } x)}$ then we can find a z in $(\text{val } (s\ x))$. The direct approach to represent "X is a value" by $\overline{(\text{val } X)}$ does not work: X is not an object. Since an object is expected at this position, X has to be converted into an object: We write $\underline{X}$ to express this conversion: The form of the second assumption is therefore $\overline{(\text{val } \underline{X})}$.

Finally, we can specify the goal, namely that the successor of X is also a value. Here we transform the program X on the object level with the projection function and apply the constant s *to construct the successor. We then have to prove that a proof object can be constructed for $\overline{(\text{val s } \underline{X})}$.*

This example motivates the definition of the projection operator $\underline{\cdot}$. It suggests, that it should be enough to restrict the domain of $\underline{\cdot}$ to meta variables. We will see, that this might not be enough: By substitution application meta variables can be instantiated with whole programs. That means an object of the form $\underline{X}$ may be instantiated to $\underline{P}$ under a substitution which replaces X by P. We examine this issue further in section 4.1. With the presence of the projection function, the diagram from above can be refined:

$$\begin{tikzcd} \text{Programs} \arrow[d, shift left, "\underline{\cdot}"] & & \text{Formulae} \\ \text{LF objects} \arrow[u, shift left, "\overline{\cdot}"] & & \text{LF types} \arrow[u, "\overline{\cdot}"'] \end{tikzcd}$$

Note that $\underline{\cdot}$ is the inverse operator to $\overline{\cdot}$ on objects of the LF level: For every LF object M we expect $\underline{(\overline{M})} = M$.

We will now define the language for kinds and types. The problem of the projection operator will be revisited when we define the language of objects. In LF types are defined as

$$\text{Types:} \quad A \quad ::= \quad a \mid (A\ M) \mid \Pi x : A_1.\, A_2$$

In the description of the the meta level, we introduced the type A_P, which is simultaneously a goal type and a data type. A_P is defined to represent atomic types. We write A_G for goal types and A_D for data types. The difference between A_G and A_D is, that no Π-types are allowed in the definition of A_G. A_G is therefore completely subsumed by A_D. We define A_G also as a set of all atomic types. A_D is defined to be either an atomic type or an Π-type:

$$\begin{array}{lll} \text{Atomic types:} & A_P & ::= \quad a \mid (A_P\ M) \\ \text{Goal types:} & A_G & ::= \quad A_P \\ \text{Data types:} & A_D & ::= \quad A_P \mid \Pi x : A_G.\, A_D \end{array}$$

Π-types can be only defined as data types. The corresponding kind must have the same parameter type as the Π-type: A_G:

$$\text{Kinds:} \quad K \quad ::= \quad \text{type} \mid \Pi x : A_G.\, K$$

The following lemma states that it is justified to call A_P, A_G and A_D types:

Lemma 3.2 (Restricted types are types) *Every atomic type is a type, every goal type is a type and every data type is a type.*

Proof: Structural induction. □

This lemma can be used to prove that G, D and C are formulae:

Lemma 3.3 (Restricted formulae are formulae) *Every goal formula is a formula, every data formula is a formula and every core formula is a formula.*

Proof: Structural Induction. Use lemma 3.2 □

We address now the definition of the language of objects: It is noteworthy to point out that the projection operator collapses the strict distinction between meta level and LF level: So far goal formulae only depend on LF types. LF types depend only on LF objects because of dependent types. LF objects can depend on programs because of the projection operator. This implies that formulae, types, and objects may depend on programs and programs may depend on objects again. Therefore, the definition of objects as

$$\text{Objects:} \quad M \quad ::= \quad \underline{P} \mid x \mid c \mid \lambda x : A_G.\, M \mid (M_1\ M_2)$$

is too general. Figure 3.1 visualizes the dependencies between objects. The dashed arrows show that programs can depend on objects, objects can depend on programs etc. The objective is

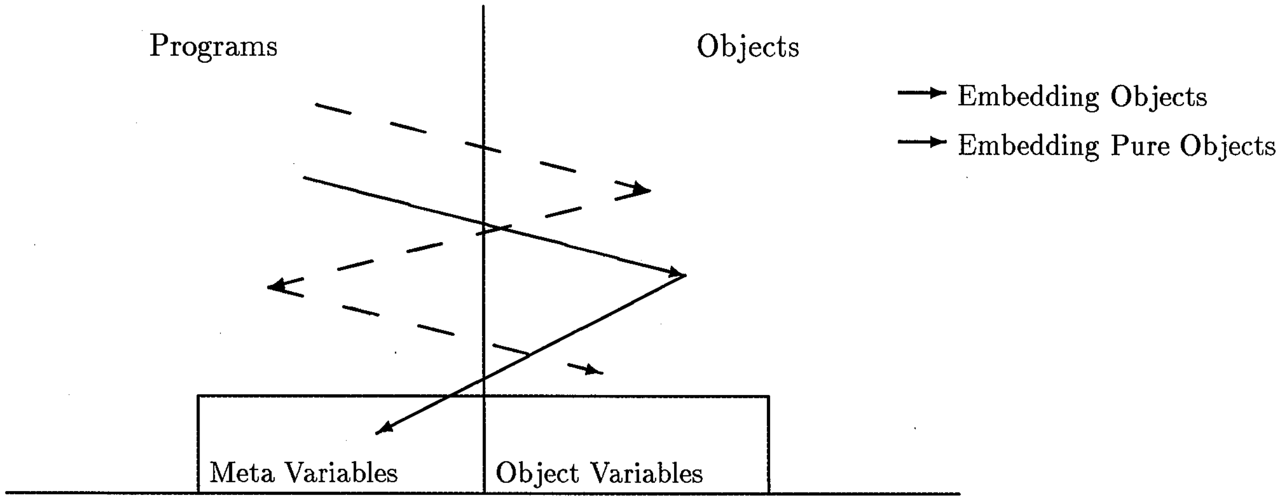

Figure 3.1: Possible Embeddings

to remove these dependencies. To restrict the dependencies, objects should only depend on program variables, not on programs by themselves. The solid arrows in figure 3.1 show this. Objects which satisfy this condition are called *pure objects*:

$$\text{Pure Objects:} \quad M \ ::= \ \underline{X} \mid x \mid c \mid \lambda x : A_G.\, M \mid (M_1\ M_2)$$

Note, that x is an object variable, defined by the LF level variable concept. The motivation for the word *pure* results from the avoidance of mutual dependencies between object and meta level. We call types A_P, A_G, A_D *pure types* and kinds K *pure kinds* if the objects on which they depend are pure objects. Programs P, which depend on pure objects all called *pure programs* and similarly formulae G, D, F, which depend on pure types are called *pure formulae.*

Lemma 3.4 *Every pure object is an object, every pure type is a type and every pure kind is a kind, every pure program is a program, every pure formula is a formula.*

Proof: follows easily from the definition. □

The distinction between pure and impure objects is not trivial. There are objects which are not pure: $eval\ \underline{(\mathtt{fun}\, X.\, X)\overline{(s\ \underline{E})}}\ \underline{V}$ for example is equivalent to $eval\ (s\ \underline{E})\ \underline{V}$ with a suitable reduction ordering. Reduction may turn non pure programs into pure ones.

In the example 3.1 we mentioned, that substitution may destroy the purity property. In the next section we develop the theory of MLF for objects — not necessarily pure — in section 4.1 we discuss the effects of restricting MLF to pure objects.

An even more restricted notion of object serves as pattern for the `case` program we introduced on the meta level. We call this patterns object patterns N. It must be prevented that during a matching operation programs with free LF variables are matched with meta variables. The matching operation will be defined in section 3.1.3. Consequently object patterns may not contain free LF variables, or λ-abstractions. We also restrict the use of constants in so far as an object pattern may only consist of *one* constant c applied to a set of projected meta variables. The formal definition is as follows:

$$\text{Object Patterns:}\quad N \;::=\; c \mid (N\ \underline{X})$$

As introduced in [HHP87, HHP93] *LF-signatures* are used to define object constants and type constants. The syntactic definition of a LF-signature is as follows:

$$\text{Signature:}\quad \Sigma \;::=\; \cdot \mid \Sigma, c : A_D \mid \Sigma, a : K$$

It is a slightly different definition from the one we introduced in section 2.2.1. From now on, we always consider Σ to be given and fixed.

3.1.1 Substitutions

In the following we introduce the concept of substitution. We are dealing with two different kind of substitutions. One kind of substitution replaces meta variables by programs — this is called a *meta level substitution* — the other object variables by objects — this is called an *object level substitution.*

We denote the empty substitution with "$\cdot$", and the constructor with ",": Meta level substitutions are denoted with Θ, object level substitutions with θ.

$$\begin{array}{lcll} \text{Meta level substitution:} & \Theta & ::= & \cdot \mid \Theta, P/X \\ \text{Object level substitution:} & \theta & ::= & \cdot \mid \theta, M/x \end{array}$$

Next, we define the union operator for substitutions, which should not be mistaken for concatenation. Let $\Theta = \cdot, P_1/X_1..P_n/X_n$ and $\Psi = \cdot, Q_1/Y_1..Q_m/Y_m$, the X_i's and the Y_j's not necessarily distinct. Then we define

$$\Theta \cup \Psi := \cdot, P_1/X_1..P_n/X_n, Q_1/Y_1..Q_m/Y_m$$

We remark that $\cup$ is not a commutative operation. This will become evident when we formalize substitution application.

The union operator is defined for two object level substitutions $\theta = \cdot, M_1/x_1..M_n/x_n$ and $\psi = \cdot, N_1/y_1..N_m/y_m$, the x_i's and the y_j's not necessarily distinct as

$$\theta \cup \psi := \cdot, M_1/x_1..M_n/x_n, N_1/y_1..N_m/y_m$$

Substitution application on the meta level has the form $[\Theta](\cdot)$, substitution application on the object level has the form $\{\theta\}(\cdot)$. From now on we omit the leading "$\cdot$," of non-empty substitutions.

Concatenation of substitutions is defined like function application. Two application of substitutions are concatenated, by applying one after the other. We have to define concatenation for meta level substitutions

Definition 3.5 (Concatenation for meta substitutions) *Let α be a program, formula, object or type. Let Θ, Ψ substitutions: We define $\Theta \circ \Psi(\alpha) := [\Theta]([\Psi](\alpha))$.*

and for object level substitutions:

Definition 3.6 (Concatenation for object substitutions) *Let α be object, type or kind. Let θ, ψ substitutions: We define $\theta \circ \psi(\alpha) := \{\theta\}(\{\psi\}(\alpha))$.*

This definition is different from the one Wayne Snyder [Sny91] uses in his book: He defines $\Theta \circ \Psi(\alpha)$ as $\Psi(\Theta(\alpha))$. Since this change of order can easily lead to confusion we decided the order to remain invariant.

Meta level substitutions

Since programs, formulae, objects, types, and kinds can depend on meta variables, the notion of substitution has to be extended to all of them. We obtain five different judgments, three of them serve to replace meta variables in objects, types, and kinds, the other two serve to replace meta variables in programs and formulae.

$$\begin{aligned}
[\Theta]_{\text{object}}(M) &= M\\
[\Theta]_{\text{type}}(A) &= A\\
[\Theta]_{\text{kind}}(K) &= K\\
[\Theta]_{\text{program}}(P) &= P\\
[\Theta]_{\text{formula}}(G) &= G
\end{aligned}$$

We first define meta level substitution on the LF level: $[\Theta]_{\text{object}}, [\Theta]_{\text{type}}$ and $[\Theta]_{\text{kind}}$. We will give the definitions in form of equations:

Definition 3.7 (Substitution on objects)

$$\begin{aligned}
[\Theta]_{object}(\underline{P}) &= \begin{cases} M & \text{if } P = X \text{ and } \Theta(X) = \overline{M}\\ \underline{[\Theta]_{program}(P)} & \text{else}\end{cases}\\
[\Theta]_{object}(x) &= x\\
[\Theta]_{object}(M_1\ M_2) &= ([\Theta]_{object}(M_1)\ [\Theta]_{object}(M_2))\\
[\Theta]_{object}(\lambda x : A.\ M) &= \lambda x : [\Theta]_{type}(A).\ [\Theta]_{object}(M)
\end{aligned}$$

We could have replaced the first equation by

$$[\Theta]_{\text{object}}(\underline{P}) = \underline{[\Theta]_{\text{program}}(P)}$$

It is easy to see, that both formulations are equivalent with respect to a conversion rule

$$\frac{}{\underline{\overline{M}} \equiv M}\ \text{Epsilon}$$

which justifies the following general equation.

$$[\Theta]_{\text{object}}(\underline{X}) = \underline{([\Theta]_{\text{program}}(X))} = \underline{\overline{M}} = M$$

Note that the conversion rule is non-standard. Under the assumption, that the programs which define Θ are normal with respect to this reduction rule, the result of the application is guaranteed to be normal, too. In this case, we do not need the conversion rule. Therefore it is not necessary to incorporate this reduction rule into MLF, as long as meta level substitution performs this reduction implicitly on objects. We will discuss this in more detail in section 3.2.

Next, we define substitution application for types. The definition is straightforward. It subsumes the definition of substitution application on the atomic, goal and data types because of lemma 3.2.

Definition 3.8 (Substitution on types)

$$\begin{array}{lcl} [\Theta]_{type}(a) & = & a \\ [\Theta]_{type}(\Pi x : A_1.\, A_2) & = & \Pi x : [\Theta]_{type}(A_1).\, [\Theta]_{type}(A_2) \\ [\Theta]_{type}(A\ M) & = & ([\Theta]_{type}(A)\ [\Theta]_{object}(M)) \end{array}$$

The substitution application on kinds is defined similarly straightforward:

Definition 3.9 (Substitution on kinds)

$$\begin{array}{lcl} [\Theta]_{kind}(\text{type}) & = & \text{type} \\ [\Theta]_{kind}(\Pi x : A.\, K) & = & \Pi x : [\Theta]_{type}(A).\, [\Theta]_{kind}(K) \end{array}$$

Now we will address the definition of meta level substitution application to programs and formulae. Since formulae can contain free meta variables, the definition of the substitution application has to be carried out with some care: the renaming of variables has to be done explicitly in all cases where variables are bound by formulae or programs:

Definition 3.10 (Substitution on formulas)

$$\begin{array}{lcl} [\Theta]_{formula}(\forall X : A.G) & = & \forall Y : [\Theta]_{type}(A).[\Theta, Y/X]_{formula}(G) \\ [\Theta]_{formula}(\exists X : A.G) & = & \exists Y : [\Theta]_{type}(A).[\Theta, Y/X]_{formula}(G) \\ [\Theta]_{formula}(G_1 \wedge G_2) & = & [\Theta]_{formula}(G_1) \wedge [\Theta]_{formula}(G_2) \\ [\Theta]_{formula}(G_1 \vee G_2) & = & [\Theta]_{formula}(G_1) \vee [\Theta]_{formula}(G_2) \\ [\Theta]_{formula}(G_1 \rightarrow G_2) & = & [\Theta]_{formula}(G_1) \rightarrow [\Theta]_{formula}(G_2) \\ [\Theta]_{formula}(1) & = & 1 \\ [\Theta]_{formula}(\overline{A}) & = & \overline{[\Theta]_{type}(A)} \end{array}$$

where Y is a new variable.

Definition 3.11 (Substitution on programs)

$$\begin{array}{lcl}
[\cdot]_{program}(X) & = & X \\
[\Theta, P/Y]_{program}(X) & = & \begin{cases} P & \textit{if } X = Y \\ [\Theta]_{program}(X) & \textit{else} \end{cases} \\
[\Theta]_{program}(\texttt{unit}) & = & (\texttt{unit}) \\
[\Theta]_{program}(\texttt{rec } X.P) & = & (\texttt{rec } Y.([\Theta, Y/X]_{program}(P))) \\
[\Theta]_{program}(\texttt{fun } X.P) & = & (\texttt{fun } Y.([\Theta, Y/X]_{program}(P))) \\
[\Theta]_{program}(\texttt{pair } P_1\ P_2) & = & (\texttt{pair } [\Theta]_{program}(P_1)\ [\Theta]_{program}(P_2)) \\
[\Theta]_{program}(\texttt{inl } P) & = & (\texttt{inl } [\Theta]_{program}(P)) \\
[\Theta]_{program}(\texttt{inr } P) & = & (\texttt{inr } [\Theta]_{program}(P)) \\
[\Theta]_{program}(\texttt{inx } P_1\ P_2) & = & (\texttt{inx } [\Theta]_{program}(P_1)\ [\Theta]_{program}(P_2)) \\
[\Theta]_{program}\begin{pmatrix} \texttt{case } P \texttt{ of} \\ \quad Q^{(1)} \Rightarrow P^{(1)} \\ \vdots \\ \mid\ Q^{(n)} \Rightarrow P^{(n)} \end{pmatrix} & = & \begin{pmatrix} \texttt{case } [\Theta]_{program}(P) \texttt{ of} \\ \quad [\Psi_1]_{program}(Q^{(1)}) \Rightarrow [\Theta \circ \Psi_1]_{program}(P^{(1)}) \\ \vdots \\ \mid\ [\Psi_n]_{program}(Q^{(n)}) \Rightarrow [\Theta \circ \Psi_n]_{program}(P^{(n)}) \end{pmatrix} \\
[\Theta]_{program}(\texttt{let } P_1 \texttt{ be } X \texttt{ in } P_2) & = & (\texttt{let } [\Theta]_{program}(P_1) \texttt{ be } Y \texttt{ in } [\Theta, Y/X]_{program}(P_2)) \\
[\Theta]_{program}(\texttt{app } P_1\ P_2) & = & (\texttt{app } [\Theta]_{program}(P_1)\ [\Theta]_{program}(P_2)) \\
[\Theta]_{program}\overline{M} & = & \overline{[\Theta]_{object}(M)}
\end{array}$$

where Y is a new variable. In the case case*, let $\{X_1^k..X_{m_k}^k\} = Free(P_2^{(k)})$ the set of free variables occuring in the pattern $P_2^{(k)}$, $\Psi_k = Y_1^k/X_1^k..Y_{m_k}^k/X_{m_k}^k$ a variable renaming substitution, where $Y_1^k..Y_{m_k}^k$ are new variable names.*

We call a substitution *pure* if all programs, which define the substitution are pure. Purity of the participating programs is not enough as we will see in section 4.1. A stronger notion is required: We call a substitution strictly pure, if its application cannot create something impure. The definition is as follows:

Definition 3.12 *(Strictly pure substitutions) A substitution Θ is called strictly pure iff for all $X \in dom(\Theta)$: $\Theta(X) = Y$ or $\Theta(X) = \overline{M}$ with M pure.*

3.1.2 LF level substitutions

LF level substitutions replace LF variables by LF objects. We assume, that programs and goals are closed with respect to object variables. Therefore we only have to define object level substitution for objects, types and kinds.

Definition 3.13 (Object substitution on kinds)

$$\begin{array}{lcl}
\{\theta\}_{kind}(type) & = & type \\
\{\theta\}_{kind}(\Pi x : A.\ K) & = & \Pi x' : \{\theta\}_{type}(A).\ \{\theta \circ (x'/x)\}_{kind}(K)
\end{array}$$

where x' is a new variable name.

Definition 3.14 (Object substitution on types)

$$\begin{array}{lcl}
\{\theta\}_{type}(a) & = & a \\
\{\theta\}_{type}(\Pi x : A_1.\, A_2) & = & \Pi x' : \{\theta\}_{type}(A_1).\, \{\theta \circ (x'/x)\}_{type}(A_2) \\
\{\theta\}_{type}(A\, M) & = & (\{\theta\}_{type}(A)\ \{\theta\}_{object}(M))
\end{array}$$

where x' is a new variable name.

Definition 3.15 (Object substitution on objects)

$$\begin{array}{lcl}
\{\cdot\}_{object}(x) & = & x \\
\{\theta, M/y\}_{object}(x) & = & \begin{cases} M & \text{if } x = y \\ \{\theta\}_{object}(x) & \text{else} \end{cases} \\
\{\theta\}_{object}(c) & = & c \\
\{\theta\}_{object}(\underline{P}) & = & P \\
\{\theta\}_{object}(\lambda x : A.\, M) & = & \lambda x' : \{\theta\}_{type}(A).\, \{\theta \circ (x'/x)\}_{object}(M) \\
\{\theta\}_{object}(M_1\, M_2) & = & (\{\theta\}_{object}(M_1)\ \{\theta\}_{object}(M_2))
\end{array}$$

where x' is a new variable name.

We observe, that the only non-standard case is the application of a substitution θ to a projected program. Because of the assumption that P is closed with respect to object variables, the result of the substitution application is P.

In the remainder of the thesis we omit the subscripts, indicating which substitution application to take: Instead of writing $[\Theta]_{\text{program}}(P)$, we write simply as $[\Theta](P)$, instead of writing $\{\theta\}_{\text{type}}(A)$ we write $\{\theta\}(A)$.

3.1.3 Unification

We define now the notion of unification. Unification will play a rôle in the definition of one of the typing rules for programs. Unification is based on meta level substitutions only.

We speak in this paragraph of terms. A term is either a program, a formula, an object, a type or a kind. A substitution Θ is called a unification of two terms, if it makes both terms syntactical equal.

Unifications can be ordered. We define the "more general relation" for substitutions: A substitution is called more general then another it the latter can be derived from the former, by further instantiation of free variables. More formally:

Definition 3.16 (More general relation:) *We say that $\Theta \leq \Psi$ iff there is a substitution Φ, s.t. $\Phi \circ \Theta = \Psi$. $\Theta \leq \Psi$ reads as Θ is more general than Ψ.*

A general observation is, that solutions of a first order unification problem are ordered with respect to the more general relation. The existence of a least element in this order is guaranteed [Sny91]. This substitution is called the most general unifier. The most common first order unification algorithms, like the Robinson algorithm or the unification algorithm of Martelli-Montanari [AM82] are guaranteed to find the most general unifier. In the higher order case, the notion of most general unifier becomes suddenly insufficient. A higher order unification problem can have arbitrary many general unifiers, which are not instantiations of each other.

Even though MLF as we introduced it in this section resembles a first order language, its connection to LF destroys the first order property. The unification problem turns out to be more complicated then simple first order unification.

We are not going into details of unification problems, the reader is referred to [Sny91]. All we need for our purposes is the notion of a unifier on the LF type level and the notion of matching on the program level. In this thesis we adopt Wayne Snyders view, of a unification problem being given as a set of equations to be solved.

Definition 3.17 (Unifier) *Let A_1, A_2 two LF types. Θ is called a unifier of A_1 and A_2 iff $\Theta(A_1) \equiv \Theta(A_2)$. We write $\Theta = \text{unify}(A_1 \approx A_2)$.*

Definition 3.18 (Matching) *Let P be a program, Q be program pattern. Θ is called a matching of P and Q iff $\Theta(P) \equiv \Theta(Q)$. We write $\Theta = \text{match}(P \approx Q)$.*

If there is a matching between a program P and a pattern Q one says that Q *matches with* P.

3.1.4 Context

For the definition of sequents which will be introduced in the next section we must provide the notion of context. As in the case of substitution we have to distinguish between two different kind of contexts. There is a context defined on the meta level which is denoted with Γ. And there is the notion of context on the LF level which is denoted with Δ.

A meta context declares meta variables as assumed proof objects for corresponding data formulae. The formulae can be seen as meta types of program variables. Note that we choose "$\in$" as a separator between a meta variable and its data formulae in order to prevent confusion with contexts as used on the LF level.

Definition 3.19 (Meta context)

$$\textit{Meta context:}\quad \Gamma \quad ::= \quad \cdot \mid \Gamma, X \in D$$

Contexts which are constructed from only pure data formulae D's are called *pure contexts*. The LF level context is defined similarly to the meta level context: Meta variables are replaced by LF variables, data formulae are replaced by data types, and the separator symbol is ":".

Definition 3.20 (LF context)

$$\textit{LF context:}\quad \Delta \quad ::= \quad \cdot \mid \Delta, x : A_D$$

We define the notion of support and the notion of free variables of meta and LF contexts: The domain of a context is a set of variable names which are introduced by the context, and the free variables of a context, are the variables which occur free in formulae and types.

Definition 3.21 (Support of a context) *Let Γ be a meta context, Δ be an LF context.*

$$\begin{aligned}&dom(\cdot) = \emptyset\\ &dom(\Gamma, X \in D) = \{X\} \cup dom(\Gamma)\\ &dom(\Delta, x : A_D) = \{x\} \cup dom(\Delta)\end{aligned}$$

The set of free variables is similarly defined:

Definition 3.22 (Free variables in a context) *Let Γ be a meta context, Δ be an LF context.*

$$\begin{aligned}&Free(\cdot) = \emptyset\\ &Free(\Gamma, X \in D) = Free(\Gamma) \cup Free(D)\\ &Free(\Delta, x : A_D) = Free(\Delta) \cup Free(A_D)\end{aligned}$$

We will not define the set of free variables for formulae, programs, types and objects. The definition is standard.

The concatenation of two contexts is written as Γ_1, Γ_2 on the meta level and as Δ_1, Δ_2 on the LF level. The overloading of the constructor "," has an advantage and a disadvantage. The disadvantage is, that it cannot be uniquely determined what "," constructs. On the other hand by using "," as context union we save new notation which makes it easier to digest the formalism introduced in the next sections and chapters. Meta context concatenation is defined by the judgement $\Gamma_1, \Gamma_2 = \Gamma_3$.

$$\frac{}{\Gamma_1, \cdot = \Gamma_1}\,\mathsf{concmetaemp} \qquad \frac{\Gamma_1, \Gamma_2 = \Gamma_3}{\Gamma_1, \Gamma_2, X \in G = \Gamma_3, X \in G}\,\mathsf{concmetanonemp}$$

LF context concatenation is defined by the judgement $\Delta_1, \Delta_2 = \Delta_3$.

$$\frac{}{\Delta_1, \cdot = \Delta_1}\,\mathsf{concobjemp} \qquad \frac{\Delta_1, \Delta_2 = \Delta_3}{\Delta_1, \Delta_2, x \in A = \Delta_3, x \in A}\,\mathsf{concobjnonemp}$$

Contexts can be also subject of substitution application. We have to introduce a new judgement: We write

$$[\Theta]_{\mathsf{context}}(\Gamma)$$

for the application of substitution to a meta context. The inference rules are as follows:

Definition 3.23 (Substitution on context)

$$\frac{}{[\Theta]_{context}(\cdot) = \cdot}\,\mathsf{substctxemp}$$

$$\frac{[\Theta]_{context}(\Gamma) = \Gamma' \quad [\Theta]_{formula}(G) = G'}{[\Theta]_{context}(\Gamma, X \in G) = \Gamma', X \in G'}\,\mathsf{substctxin} \quad X \notin dom(\Theta)$$

$$\frac{[\Theta]_{context}(\Gamma) = \Gamma'}{[\Theta]_{context}(\Gamma, X \in G) = \Gamma'}\,\mathsf{substctxnotin} \quad X \in dom(\Theta)$$

This definition may seem a little peculiar: Substitution application on contexts is not performed in a declaration by declaration manner, but meta variable declarations can be removed. Θ must be seen as a refinement substitution, that is variables in $Free(\Theta)$ are refined by its application. When all free occurrences of a variable X are removed, then the declaration of X in the context is unnecessary. Finally, we define the application of meta level substitutions to LF contexts:

Definition 3.24 (Substitution on object context)

$$\frac{}{[\Theta]_{objctx}(\cdot) = \cdot}\,\textsf{substobjctxemp}$$

$$\frac{[\Theta]_{objctx}(\Delta) = \Delta' \quad [\Theta]_{formula}(A) = A'}{[\Theta]_{objctx}(\Delta, x : A) = \Delta', x : A'}\,\textsf{substobjctxnonemp}$$

It is easy to see that substitution has the following property: Extracting typing information from a context and substitution application are associative. Here is the lemma without proof:

Lemma 3.25

$$\begin{aligned} [\Theta]_{context}(\Gamma)(X) &= [\Theta]_{formula}(\Gamma(X)) && (3.1)\\ [\Theta]_{objctx}(\Delta)(x) &= \{\Theta\}_{type}(\Delta(x)) && (3.2) \end{aligned}$$

In the next section we will define typing rules, which distinguish between well-formed and ill-formed contexts.

3.2 Reduction relation and Evaluation

In this section we state the reduction relation of LF objects, LF types, and LF kinds. We also propose a reduction relation for programs. As remarked earlier, programs can be seen as an extension of the simply-typed λ-terms. Hence, β and η reduction are reduction rules. For the other operational programs, we define more reduction rules. From the theory of the λ-calculus it is well-known, that λ-terms might not reduce to normal forms. Evaluation orderings are defined which motivates the notion of canonical forms. At the end of this section we define an inference rule system for an evaluation judgment based on the eager evaluation ordering.

3.2.1 Reduction Relation for LF Objects, LF Types, and LF Kinds

In the original definition of the logical framework LF, a congruence relation between objects, types and kinds is defined. We denote the congruence relation between kinds with $K_1 \equiv K_2$, between types with $AP_1 \equiv AP_2$, $AG_1 \equiv AG_2$, $AD_1 \equiv AD_2$ and between objects between as $M_1 \equiv M_2$. In the next subsection we present a reduction relation for programs. Since projected programs are objects, there is a mutual dependency between programs and LF objects, which must be expressed in the definition of the reduction relation. This dependency destroys the clean distinction between LF and meta level. This is why we decided to restrict the reduction relation on programs to syntactical identity. This way, we manage to preserve the clean distinction.

The notion of reduction is necessary because in the simply typed λ-calculus, λ-terms are to be considered equal if they are either β-reducible or η-reducible to each other. This observation applies to the level of LF objects. We have the following two rules:

$$\frac{}{(\lambda x : A_G.\ M)\ M' \equiv \{\cdot, N/x\}(M')}\ \textsf{objbeta}$$

$$\frac{}{(\lambda x : A_G.\ (M\ x)) \equiv M}\ \textsf{objeta}$$

We observe that even so $\equiv$ is defined only on LF object level, it will have effects on types and on kinds. Types and kinds are depending on objects, so it is necessary to define a notion of equivalence on types and kinds. We can define a set of rules which ensures reflexivity, symmetry, and transitivity of $\equiv$. We are not giving the rules here, the reader is referred to [HHP93]. At last we have to make sure that $\equiv$ is a congruence relation. We do not give the rules for kinds and types, they remain as described in [HHP93]. The set of rules for objects must be extended, due to the presence of projected programs or meta variables. The following rules remain unchanged:

$$\frac{A_G \equiv A_G{}'}{\lambda x : A_G.\ M \equiv \lambda x : A_G{}'.\ M}\ \textsf{objlamA}$$

$$\frac{M \equiv M'}{\lambda x : A_G.\ M \equiv \lambda x : A_G.\ M'}\ \textsf{objlamB}$$

$$\frac{M_1 \equiv M_1'}{M_1\ M_2 \equiv M_1'\ M_2}\ \textsf{objappA}$$

$$\frac{M_2 \equiv M_2'}{M_1\ M_2 \equiv M_1\ M_2'}\ \textsf{objappB}$$

The new rule is the rule which reduces projected programs. If an object is constructed by the projection of a program onto the LF level, only syntactical identical programs are considered to be equivalent. In the case of pure objects, this is no restriction: pure objects only depend on projected meta variables:

$$\frac{}{\underline{X} \equiv \underline{X}}\ \textsf{objprg}$$

The reduction rule in the impure case has therefore the form:

$$\frac{}{\underline{P} \equiv \underline{P}}\ \textsf{objprg} \qquad (3.3)$$

3.2.2 Reduction Relation for programs

We know from the λ-calculus that syntactically different λ-terms are considered to be semantically equal. This notion is made more precise by a so called reduction relation. λ-terms can be rewritten using reduction rules. β-reduction and η-reduction are the only two rules defined for the λ-calculus. Programs are very similar to λ-terms. The program ($\texttt{fun}\ X.P$) corresponds to λ-abstraction. In this section we will restate β and η reduction for programs, and extend the set of reduction rules for other programs. Note, that we omit all typing information from the programs. We can do that, because we can assume the programs always to be well-typed. The typing rules for programs are defined in the next section. The second assumption is, that programs do not contain any occurrences of free meta variables.

Under this general assumption we define now the inference rules for the reduction relation for programs: $P \equiv P$. We define reduction rules for application, function, case distinction, assignment, embedding and recursion.

β-reduction

The first rule is β-reduction. If a function is applied to a program Q, it can be reduced to body of the function by replacing the bound variable by Q.

$$\frac{}{(\texttt{fun}\ X.P)\ Q \equiv [Q/X](P)}\ \textsf{Beta}$$

η-reduction

The second rule corresponds to η-reduction. The program — which represents a function abstraction where the body is the application of a function F to the newly bound variable — can be reduced to F.

$$\frac{}{(\texttt{fun}\ Y.(F\ Y)) \equiv F}\ \textsf{Eta}$$

γ-reduction

The third rule is a rule which reduces case expressions. We call this reduction γ-reduction. Assume a case program is given, which first parameter is of the form $\overline{c_i\ \underline{Q_1}..\underline{Q_{m_i}}}$. Operationally speaking the program defined on the i-th branch of the case command has to be executed after an appropriate variable substitution. Note, that the number of parameters must be identical to the number of variable slots, provided by the pattern. The reduction rule is defined as follows: The matching substitution between pattern and $\overline{c_i\ \underline{Q_1}..\underline{Q_{m_i}}}$ has the form: $Q_1/X_1..Q_{m_i}/X_{m_i}$. This substitution binds the variables in the program $P^{(i)}$ to the new values.

$$\frac{}{\left(\begin{array}{l} \textsf{case}\ \overline{c_i\ \underline{Q_1}..\underline{Q_{m_i}}}\ \textsf{of} \\ \quad \overline{c_1\ \underline{X_1^{(1)}}..\underline{X_{m_1}^{(1)}}} \Rightarrow P^{(1)} \\ \quad \vdots \\ \mid \quad \overline{c_n\ \underline{X_1^{(n)}}..\underline{X_{m_n}^{(n)}}} \Rightarrow P^{(n)} \end{array}\right) \equiv [Q_1/X_1..Q_{m_i}/X_{m_i}](P^{(i)})}\ \textsf{Gamma}$$

λ-reduction

The fourth reduction rule is the assignment reduction. We call it λ-reduction. The `let` construct takes a parameter program P_1, binds it to a variable X and replaces all free occurrences of X in P_2 with P_1. The rule has the following form.

$$\frac{}{(\texttt{let}\ P_1\ \texttt{be}\ X\ \texttt{in}\ P_2) \equiv [P_1/X](P_2)}\ \textsf{Lambda}$$

ϵ-reduction

The fifth reduction rule is called ϵ-reduction. It is only important in the case of impure programs. It states, that every immediate pair of projection and embedding can be removed:

$$\frac{}{\overline{\underline{P}} \equiv P}\ \textsf{Epsilon}$$

ρ-reduction

The sixth and last reduction rule treats recursion: it is called the ρ-reduction. All free occurrences of the variable X are replaced by the recursive program itself.

$$\frac{}{(\texttt{rec}\ X.\ P) \equiv [\texttt{rec}\ X.\ P/X](P)}\ \textsf{Roh}$$

It would be beyond the scope of this thesis to go into a detailed examination about the character and theoretical properties of this set of reduction rules. For an arbitrary program, these reduction rules have a very non-deterministic flavor. We conjecture, that the reduction of a well-typed program stops after finite many applications.

3.2.3 Design of an MLF Evaluation Function

In the last subsection we defined a set of reduction rules, which can reduce subterms of a term at any time. In this subsection we define an evaluation judgment for MLF. The application of reduction rules is triggered by the form of the program. We have seen that some of the reduction rules make use of substitutions. The substitutions are constructed from the form of the program — as for example for application. There are two common strategies how to apply reduction rules. The first evaluation strategy is called *eager evaluation*. The programs which are used for the definition of substitutions are evaluated before the substitution is formed. The second evaluation strategy is called *lazy evaluation*. Programs are not evaluated, but taken directly to form the substitution. In this thesis we follow only the ideas of the first strategy: The judgment we define for the eager evaluation strategy of MLF program has the form:

$$P \hookrightarrow_{MLF} P$$

In the remainder of this section, we define the inference rules for this judgement: Reduction rules are applied in an outermost left to right fashion. We give the rules by distinguishing between the forms of the program P in the judgment $P \hookrightarrow_{MLF} Q$:

The first rule says that the program, which is the proof term for 1 evaluates to itself.

$$\frac{}{\texttt{unit} \hookrightarrow_{MLF} \texttt{unit}}\ \textsf{evunit}$$

Another program which evaluates to itself is the λ-term — or in our notation the term ($\texttt{fun}\ X.P$):

$$\frac{}{\texttt{fun}\ X.P \hookrightarrow_{MLF} \texttt{fun}\ X.P}\ \textsf{evfun}$$

These are the only base rules for the evaluation judgement. We will now address the other programs: The recursion program corresponds to the fixed point construct of a functional programming language. The definition of the evaluation rule is defined similarly to the ρ-reduction:

$$\frac{[\texttt{rec}\ X.P/X](P) \hookrightarrow_{MLF} V}{\texttt{rec}\ X.P \hookrightarrow_{MLF} V}\ \textsf{evrec}$$

The rules for the evaluation of `pair`, `inl`, `inr` and `inx` are all very similar, the definition is straightforward.

$$\frac{P_1 \hookrightarrow_{MLF} V_1 \quad P_2 \hookrightarrow_{MLF} V_2}{(\texttt{pair}\ P_1\ P_2) \hookrightarrow_{MLF} (\texttt{pair}\ V_1\ V_2)}\ \textsf{evpair}$$

$$\frac{P \hookrightarrow_{MLF} V}{(\texttt{inl}\ P) \hookrightarrow_{MLF} (\texttt{inl}\ V)}\ \textsf{evinl}$$

$$\frac{P \hookrightarrow_{MLF} V}{(\texttt{inr}\ P) \hookrightarrow_{MLF} (\texttt{inr}\ V)}\ \textsf{evinr}$$

$$\frac{P_1 \hookrightarrow_{MLF} V_1 \quad P_2 \hookrightarrow_{MLF} V_2}{(\texttt{inx}\ P_1\ P_2) \hookrightarrow_{MLF} (\texttt{inx}\ V_1\ V_2)}\ \textsf{evinx}$$

The next evaluation judgment defines the evaluation of a case program. The idea is to evaluate the first parameter. The result is supposed to be of the form: $\overline{c_i\ \underline{Q_1}..\underline{Q_{m_i}}}$. If the case construct includes the definition of a pattern which matches this program, the evaluation is possible: In this case, the i-th program $P^{(i)}$ is selected, the variables are instantiated with the programs $Q_1..Q_{m_i}$, and then evaluated. The result value is the value of the evaluation of the case construct.

$$\frac{P \hookrightarrow_{MLF} \overline{c_i\ \underline{Q_1}..\underline{Q_{m_i}}} \quad [Q_1/X_1..Q_{m_i}/X_{m_i}](P^{(i)}) \hookrightarrow_{MLF} V}{\left(\begin{array}{l} \textsf{case}\ P\ \textsf{of} \\ \quad \overline{c_1\ \underline{X_1^{(1)}}..\underline{X_{m_1}^{(1)}}} \Rightarrow P^{(1)} \\ \quad \vdots \\ \mid \quad \overline{c_n\ \underline{X_1^{(n)}}..\underline{X_{m_n}^{(n)}}} \Rightarrow P^{(n)} \end{array}\right) \hookrightarrow_{MLF} V}\ \textsf{evcase}$$

To evaluate a `let` construct, we first evaluate the program P_1 to obtain a value V'. We then replace X in P_2 by V'. The evaluation of $[V'/X](P_2)$ yields the value V. This rule is clearly eager because the program P_1 is evaluated *before* it is substituted into P_2.

$$\frac{P_1 \hookrightarrow_{MLF} V' \quad [V'/X](P_2) \hookrightarrow_{MLF} V}{(\texttt{let}\ P_1\ \texttt{be}\ X\ \texttt{in}\ P_2) \hookrightarrow_{MLF} V}\ \mathsf{evleteager}$$

We also can imagine a lazy version of this rule:

$$\frac{[P_1/X](P_2) \hookrightarrow_{MLF} V}{(\texttt{let}\ P_1\ \texttt{be}\ X\ \texttt{in}\ P_2) \hookrightarrow_{MLF} V}\ \mathsf{evletlazy}$$

The application rule is similar to the `let` rule. Both rules are closely related. It is even possible to replace the `let` program by an application and a `fun`-abstraction. We can define `let` as syntactic sugar from `fun`:

$$(\texttt{let}\ P_1\ \texttt{be}\ X\ \texttt{in}\ P_2) \equiv ((\texttt{fun}\ X.P_2)\ P_1)$$

Since we want to keep proof terms readable, we decided to use the `let` construct instead of the application. The rule for eager application is defined as

$$\frac{P_1 \hookrightarrow_{MLF} (\texttt{fun}\ X.P') \quad P_2 \hookrightarrow_{MLF} V' \quad [V'/X](P') \hookrightarrow_{MLF} V}{(\texttt{app}\ P_1\ P_2) \hookrightarrow_{MLF} V}\ \mathsf{evappeager}$$

Again it is possible to write down the lazy form: P_2 is not evaluated, but directly substituted into P':

$$\frac{P_1 \hookrightarrow_{MLF} (\texttt{fun}\ X.P') \qquad [P_2/X](P') \hookrightarrow_{MLF} V}{(\texttt{app}\ P_1\ P_2) \hookrightarrow_{MLF} V}\ \mathsf{evapplazy}$$

To complete the inference rule set for the evaluation judgment we define the evaluation of embedded LF objects. If we can assume all programs to be pure, it has the following form:

$$\frac{}{\overline{M} \hookrightarrow_{MLF} \overline{M}}\ \mathsf{evobj}$$

M cannot contain any free variables, all evaluation rules preserve the property, that the programs do not contain any free variables. The only variables which can occur free in M are free meta variables. It follows immediately, that M does not contain any programs at all.

For the impure case, things are getting much more difficult: The object structure of M has to be examined because an unevaluated program could be a subformula of M. The rule has then the following form:

$$\frac{M_1 \equiv M_2}{\overline{M_1} \hookrightarrow_{MLF} \overline{M_2}}\ \mathsf{evobj}$$

But then the rule (3.3) has to be exchanged by the new rule:

$$\frac{P_1 \equiv P_2}{\underline{P_1} \equiv \underline{P_2}}\ \mathsf{objprg}'$$

This is not really satisfactory. The aim of MLF is to reason about pure objects only.

Definition 3.26 (Canonical form) *Let P be a well-formed program which does not contain any free meta variables. If $P \hookrightarrow_{MLF} P'$, P' is called the canonical form of P.*

3.3 MLF inference system

In this section we introduce the typing judgments and typing rules for MLF. In the last section we defined several languages for formulae, programs, kinds, types and objects. In this section we combine these notions, and describe the different typing relations. Therefore we define a collection of new judgments. The semantics of these judgments is defined in the following subsections. Since the typing rules for each judgments will depend on other judgments, we first define the judgments and describe their meaning informally before we go into details. The judgments have the following forms:

1.	$\vdash_\Sigma \Gamma$ ctx		
2.	$\Gamma \vdash_\Sigma G$ goal	$\Gamma \vdash_\Sigma D$ data	
3.	$\Gamma \vdash_\Sigma P \in G$		
4.	$\Gamma \vdash_\Sigma \Delta$ objctx		
5.	$\Gamma; \Delta \vdash_\Sigma K$ kind		
6.	$\Gamma; \Delta \vdash_\Sigma A_P : K$	$\Gamma; \Delta \vdash_\Sigma A_G : K$	$\Gamma; \Delta \vdash_\Sigma A_D : K$
7.	$\Gamma; \Delta \vdash_\Sigma M : A_P$	$\Gamma; \Delta \vdash_\Sigma M : A_G$	$\Gamma; \Delta \vdash_\Sigma M : A_D$

The first judgement allows us to distinguish between well-formed and ill-formed contexts. The inference of this judgment will reflect that in a context $\Gamma, X \in D$, D may only depend on free variables in Γ. If this holds for every declaration in a context, then we call the context well-formed otherwise ill-formed.

The second set of judgements $\Gamma \vdash_\Sigma G$ goal and $\Gamma \vdash_\Sigma D$ data express the property if a formula G is a well-formed goal formula and the formula D is a well-formed data formula. Since formulae can depend on free variables, the judgments must contain the context Γ.

The third judgement is the center piece of the MLF rule system. It defines the typing relation between programs and formulae. This judgement can be seen from two different angles: from a logical angle and from a computer science angle.

From a logical point of view, the context represents the set of assumptions, from which the formulae G has to be proven. The proof term P represents the derivation.

From a computer science point of view, the judgment $\Gamma \vdash_\Sigma P \in G$ represents a state in a computation. The context Γ accounts for all objects available at this certain stage. All objects are disguised in form of programs. P stands for the program still to be executed. G is the "type" of the result. Let $X \in \overline{A}$ be a declaration in the context, A is an LF type. One can expect that X is bound to a value $\overline{M}$ at this stage of the computation. In the last section of this chapter we will introduce an evaluation judgment which actually calculates the result program from given programs.

The fourth judgment $\Gamma \vdash_\Sigma \Delta$ objctx defines if an LF context Δ is well-formed with respect to a meta context Γ. The meta context has to be part of the judgment, since the declarations in Δ may depend on meta variables.

The next three sets of judgments redefine the judgments of LF type theory. Since MLF is built on top of the Horn fragment of there are three different types A_P, A_G and A_D. For each type there must be a typing judgment which defines the kind of the type. There must be also a typing judgment to define how the objects of this type may look like.

This section is organized as follows: We first define the rules for well-formed contexts. We then show the revised versions of the judgments for LF type theory, that is what are well-formed objects, what are well-formed types and what are well-formed kinds. Finally we define the inference rules for MLF: What are well-formed data formulae, what are well-formed goal formulae, and what are well-formed programs.

3.3.1 Typing rules for meta context

In this subsection we define the inference rules for the judgment $\vdash_\Sigma \Gamma$ ctx: Is Γ a well-formed meta context? A meta context is well-formed if it is either empty or all declarations can be shown to be well-formed. A declaration is well-formed, if D is a well-formed data formula:

$$\frac{}{\vdash_\Sigma \cdot \text{ ctx}}\,\textsf{ctxemp} \qquad \frac{\vdash_\Sigma \Gamma \text{ ctx} \quad \Gamma \vdash_\Sigma D \text{ data}}{\vdash_\Sigma \Gamma, X \in D \text{ ctx}}\,\textsf{ctxcons}$$

3.3.2 Typing rules for object contexts

In this subsection we give the rules to derive the judgment $\Gamma \vdash_\Sigma \Delta$ objctx for well-typed object contexts. This judgement is essentially the same judgment as it is described in the original paper [HHP93]. We have to refine it because objects can depend meta variables. The idea is to formulate the judgement in a way, that Δ is a well-formed context with respect to meta variables in Γ. As in the meta context definition we have to check, whether every entry $x : A_D$ in Δ is defined with respect to a given Γ.

$$\frac{}{\Gamma \vdash_\Sigma \cdot \text{ objctx}}\,\textsf{objctxemp}$$

$$\frac{\Gamma \vdash_\Sigma \Delta \text{ objctx} \quad \Gamma; \Delta \vdash_\Sigma A_D : \text{type}}{\Gamma \vdash_\Sigma \Delta, x : A_D \text{ objctx}}\,\textsf{objctxcons}$$

3.3.3 Typing rules for kinds

In this subsection we give the rules to derive the judgment $\Gamma; \Delta \vdash_\Sigma K$ kind for kinds. We take the judgment from LF type theory and extend it by the context Γ. The rules are as follows:

$$\frac{}{\Gamma; \Delta \vdash_\Sigma \text{type kind}}\,\textsf{kindtype}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma A_G : \text{type} \quad \Gamma; \Delta, x : A_G \vdash_\Sigma K \text{ kind}}{\Gamma; \Delta \vdash_\Sigma \Pi x : A_G.\, K \text{ kind}}\,\textsf{kindpi}$$

Note, that every A_G is also an A_D.

3.3.4 Typing rules for types

In this subsection we give the rules to derive the judgment for types. As we have seen earlier, we distinguish between atomic types, goal types and data types. We introduced three judgments:

$$\Gamma; \Delta \vdash_\Sigma A_P : K$$
$$\Gamma; \Delta \vdash_\Sigma A_G : K$$
$$\Gamma; \Delta \vdash_\Sigma A_D : K$$

As we know from LF type theory, types are *typed* by kinds. We already know how to judge about kinds. Note that there is no direct connection between MLF and LF in these rules. Types do not depend neither on programs nor on formulae. Indirectly, they may depend on programs, since objects depend on programs and types may depend on objects. We will be concerned with this question in the next subsection. We take the judgment from LF type theory and extend it by the context Γ. The rules for all three judgments have the form:

$$\frac{\Sigma(a) = K}{\Gamma; \Delta \vdash_\Sigma a : K}\ \textsf{typeatomconst}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma A_P : \Pi x : A_G.\, K \quad \Gamma; \Delta \vdash_\Sigma M : A_G}{\Gamma; \Delta \vdash_\Sigma (A_P\ M) : \{M/x\}_{\text{kind}}(K)}\ \textsf{typeatomapp}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma A_P : K \quad K \equiv K' \quad \Gamma; \Delta \vdash_\Sigma K' : \text{kind}}{\Gamma; \Delta \vdash_\Sigma A_P : K'}\ \textsf{typeatomequiv}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma A_G : \text{type} \quad \Gamma; \Delta, x : A_G \vdash_\Sigma A_D : \text{type}}{\Gamma; \Delta \vdash_\Sigma \Pi x : A_G.\, A_D : \text{type}}\ \textsf{typedatapi}$$

3.3.5 Typing rules for objects

In this subsection we give the rules to derive the typing judgments for objects:

$$\Gamma; \Delta \vdash_\Sigma M : A_P$$
$$\Gamma; \Delta \vdash_\Sigma M : A_G$$
$$\Gamma; \Delta \vdash_\Sigma M : A_D$$

In LF type theory there is judgment $\Delta \vdash_\Sigma M : A$. Since there are atomic, goal and data types, a set of rules has to be defined for each judgment. The judgments are extended by the meta context Γ.

$$\frac{\Delta(x) = A_D}{\Gamma; \Delta \vdash_\Sigma x : A_D}\ \textsf{objdatasigma}$$

$$\frac{\Sigma(c) = A_D}{\Gamma; \Delta \vdash_\Sigma c : A_D}\ \textsf{objdataconst}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma M_1 : \Pi x : A_G.\, A_D \quad \Gamma; \Delta \vdash_\Sigma M_2 : A_G}{\Gamma; \Delta \vdash_\Sigma (M_1\, M_2) : A_D}\,\mathsf{objdataapp}$$

$$\frac{\Gamma; \Delta, x : A_G \vdash_\Sigma M : A_D}{\Gamma; \Delta \vdash_\Sigma \lambda x : A_G.\, M : \Pi x : A_G.\, A_D}\,\mathsf{objdatapi}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma M : A_D \quad A_D \equiv A_D{}' \quad \Gamma; \Delta \vdash_\Sigma A_D{}' : \text{type}}{\Gamma; \Delta \vdash_\Sigma M : A_D{}'}\,\mathsf{objdataequiv}$$

Earlier, we introduced two different possibilities of how to form objects from programs. In the general case — an object is formed by projecting a program onto the object level, in the restricted case, the form of the programs is restricted to variables. We called objects of the latter form pure objects.

The typing rule for the impure case has the following form: If P is a proof term of a formula G and G is of the form $\overline{A}$ then $\underline{P}$ is of type A:

$$\frac{\Gamma \vdash_\Sigma P \in \overline{A_G}}{\Gamma; \Delta \vdash_\Sigma \underline{P} : A_G}\,\mathsf{objgoalprgl} \tag{3.4}$$

If we read the rule from bottom to top, we read it as: If $\underline{P}$ should be shown of type A in context Δ — the meta variables are all defined in a context Γ, then P has to be shown to be a program of formula $\overline{A}$, solely from the context Γ. We assume that every program is closed with respect to LF variables. Note, that with this rule it is possible to generate cyclic dependencies between LF level and meta level. These dependencies must be removed since the goal is to obtain a clean distinction between LF and meta level. A simplification arises from assuming that the objects in question are actually pure. That is, only meta variables can be projected onto the LF level and not arbitrary programs any more. We obtain a simplified typing rule:

$$\frac{\Gamma(X) = \overline{A_G}}{\Gamma; \Delta \vdash_\Sigma \underline{X} : A_G}\,\mathsf{objgoalprgP} \tag{3.5}$$

This formulation of the rule removes the mutual dependencies between LF and meta level.

3.3.6 Typing rules for formulae

The judgments

$$\Gamma \vdash_\Sigma G \text{ goal}$$
$$\Gamma \vdash_\Sigma D \text{ data}$$

define the well-formedness of goal formulae and data formula. It is easy to see, that core formulae are well-formed if and only it is well-formed as a goal formula if and only if it is well-formed as a data formula. Goal formula and data formulae may depend on free meta variables. Hence, the judgments depend on the context Γ. The rules for goal formulae are defined in a straightforward manner:

$$\frac{\Gamma; \cdot \vdash_\Sigma A_D : \text{type} \quad \Gamma, X \in \overline{A_D} \vdash_\Sigma G \text{ goal}}{\Gamma \vdash_\Sigma \forall X : A_D.G \text{ goal}} \textsf{goalforall}$$

$$\frac{\Gamma; \cdot \vdash_\Sigma A_G : \text{type} \quad \Gamma, X \in \overline{A_G} \vdash_\Sigma G \text{ goal}}{\Gamma \vdash_\Sigma \exists X : A_G.G \text{ goal}} \textsf{goalexists}$$

$$\frac{\Gamma \vdash_\Sigma G_1 \text{ goal} \quad \Gamma \vdash_\Sigma G_2 \text{ goal}}{\Gamma \vdash_\Sigma G_1 \wedge G_2 \text{ goal}} \textsf{goaland}$$

$$\frac{\Gamma \vdash_\Sigma G_1 \text{ goal} \quad \Gamma \vdash_\Sigma G_2 \text{ goal}}{\Gamma \vdash_\Sigma G_1 \vee G_2 \text{ goal}} \textsf{goalor}$$

$$\frac{\Gamma \vdash_\Sigma D \text{ data} \quad \Gamma \vdash_\Sigma G \text{ goal}}{\Gamma \vdash_\Sigma D \rightarrow G \text{ goal}} \textsf{goalimp}$$

$$\frac{}{\Gamma \vdash_\Sigma 1 \text{ goal}} \textsf{goaltrue}$$

$$\frac{\Gamma; \cdot \vdash_\Sigma K \text{ kind} \quad \Gamma; \cdot \vdash_\Sigma A_G : K}{\Gamma \vdash_\Sigma \overline{A_G} \text{ goal}} \textsf{goaltype}$$

The rules for data formulae are similarly defined as:

$$\frac{\Gamma; \cdot \vdash_\Sigma A_G : \text{type} \quad \Gamma, X \in \overline{A_G} \vdash_\Sigma D \text{ data}}{\Gamma \vdash_\Sigma \forall X : A_G.D \text{ data}} \textsf{dataforall}$$

$$\frac{\Gamma; \cdot \vdash_\Sigma A_D : \text{type} \quad \Gamma, X \in \overline{A_D} \vdash_\Sigma D \text{ data}}{\Gamma \vdash_\Sigma \exists X : A_D.D \text{ data}} \textsf{dataexists}$$

$$\frac{\Gamma \vdash_\Sigma D_1 \text{ data} \quad \Gamma \vdash_\Sigma D_2 \text{ data}}{\Gamma \vdash_\Sigma D_1 \wedge D_2 \text{ data}} \textsf{dataand}$$

$$\frac{\Gamma \vdash_\Sigma D_1 \text{ data} \quad \Gamma \vdash_\Sigma D_2 \text{ data}}{\Gamma \vdash_\Sigma D_1 \vee D_2 \text{ data}} \textsf{dataor}$$

$$\frac{\Gamma \vdash_\Sigma G \,\text{goal} \quad \Gamma \vdash_\Sigma D \,\text{data}}{\Gamma \vdash_\Sigma G \to D \,\text{data}}\ \textsf{dataimp}$$

$$\frac{}{\Gamma \vdash_\Sigma 1 \,\text{data}}\ \textsf{datatrue}$$

$$\frac{\Gamma; \cdot \vdash_\Sigma K \,\text{kind} \quad \Gamma; \cdot \vdash_\Sigma A_D : K}{\Gamma \vdash_\Sigma \overline{A_D} \,\text{data}}\ \textsf{datatype}$$

3.3.7 Typing rules of MLF

The central notion for meta level reasoning is the sequent. A sequent represents information about a context, a goal formula which is to be proven, and a proof term. The context represents also variable dependencies. A sequent is of the form $\Gamma \vdash_\Sigma P \in G$.

The typing rules of programs in MLF are designed in sequent calculus style. We distinguish between left and right rules, that is rules which operate on the context and rules, which operate on the goal formula. The calculus represents essentially intuitionistic first order logic. Two non-standard rules are added to the system. One rule is the recursion rule: it provides — in the case of an induction proof the appropriate induction hypothesis. The well-foundedness of the recursion is *not* incorporated in the system, but encoded in form of a side condition. This approach has two advantages. First, we get a cleaner inference rule system and second a more powerful proof system, because the well-foundedness proofs have to be done outside this system. The second advantage is, that different methods can now be used to prove well-foundedness as a property of the proof term.

The second rule added to the system is a case distinction rule. This rule allows to differentiate over different forms of an LF object. Note, that since the signature is finite, only finite many cases have to be considered. If an LF object M of LF type A is given, all possible forms of M as an element of A are examined. A variable refinement substitution is derived by pattern matching. This substitution accounts for the dependencies of newly introduced variables to old ones.

The judgment $\Gamma \vdash_\Sigma P \in G$ reads as follows. Γ represents the context — that is it defines all meta variables which may occur free in P and in G. G is the goal formula to be proven. Once a proof is found, a proof term P is available. This proof term can be read as a functional program with patterns. We will first present the axiom cases, the right rules, the recursion rule, then the left rules, the case distinction rule, and finally the cut rule.

Axioms

The first axiom rule is purely logical. It simply says, that if we have a proof of goal C in the context, then we have a proof of goal C.

$$\frac{\vdash_\Sigma \Gamma_1, X \in C, \Gamma_2 \,\text{ctx}}{\Gamma_1, X \in C, \Gamma_2 \vdash_\Sigma X \in C}\ \textsf{id}$$

The second axiom rule is similarly simple. MLF is a meta logical system on top of LF. We assume to work in a fixed signature Σ. The embedding functions we defined earlier allow LF objects to serve as proof terms — proof terms for embedded LF types as MLF formulae. The constant rule allows MLF to access constants defined in the signature:

$$\frac{\vdash_\Sigma \Gamma \text{ ctx}}{\Gamma \vdash_\Sigma \overline{c} \in \overline{A_G}}\,\mathsf{const} \qquad \text{for } c : A_G \text{ defined in } \Sigma$$

The third axiom rule can be seen as a logical rule, too. In some theorem provers, the formula "true" is represented as a formula like $A \vee \neg A$. This representation avoids the definition of a new constant, and the corresponding inference rules. We do not follow this idea. "True" is defined as a formula 1 in section 3.1. Therefore we have to define an inference rule and a proof term for 1. In every arbitrary context Γ, `unit` is a proof for the formula 1.

$$\frac{\vdash_\Sigma \Gamma \text{ ctx}}{\Gamma \vdash_\Sigma (\texttt{unit}) \in 1}\,\mathsf{R1}$$

We will now describe the set of right rules which are very similar to the right of the sequent calculus for intuitionistic first order logic.

Right Rules:

Right rules operate on the goal formula — the only formula on the right hand side of a sequent. A right rule can be applied in a bottom to top manner to resolve the structure of the goal, and generate a set of subgoals to be proven. Right rules are always applicable if the goal formulae is of composite form, that is it is not atomic. We define rules for five intuitionistic connectives: conjunction, disjunction, implication, universal quantification, and existential quantification.

The rule for conjunction on the right is straightforward. A goal $G_1 \wedge G_2$ can be proven if G_1 and G_2 can be proven independently. The resulting proof term is a pair of both proofs. We introduce `pair` as a constructor for pairs in programs.

$$\frac{\Gamma \vdash_\Sigma P_1 \in G_1 \quad \Gamma \vdash_\Sigma P_2 \in G_2}{\Gamma \vdash_\Sigma (\texttt{pair}\ P_1\ P_2) \in G_1 \wedge G_2}\,\mathsf{R}\wedge$$

The rule for disjunction is defined as in the intuitionistic case. We can prove the disjunction of G_1 and G_2 if and only if at least G_1 or G_2 are provable. The proof term is constructed by `inr` or `inl` and the proof term of the premiss as argument. The proof term has to store the information if the left or the right side of the goal has been proven. That's why we have to distinguish two constructors.

$$\frac{\Gamma \vdash_\Sigma P \in G_1}{\Gamma \vdash_\Sigma (\texttt{inl}\ P) \in G_1 \vee G_2}\,\mathsf{R}\vee_1 \qquad \frac{\Gamma \vdash_\Sigma P \in G_2}{\Gamma \vdash_\Sigma (\texttt{inr}\ P) \in G_1 \vee G_2}\,\mathsf{R}\vee_2$$

The definition of the rule for implication is also very similar to the intuitionistic case. As described in [GLT88] a very nice way to represent proofs in the intuitionistic calculus is to make use of the Curry Howard isomorphism, and to present the proofs as λ-terms. The proof term of an implication on the right is a λ-term. This term essentially reads as: From a proof of D, a proof for G can be constructed. We restrict the left side of implication to data formulae and the right side to goal formulae because we are working in the Horn fragment of LF. Since we want to make a clear distinction between λ-terms as they are used in LF type theory on the object level, we use a different notion to represent λ-terms: (fun $X.P$) corresponds to a λ-term on program level.

The rule for implication on the right reads as follows: If $D \to G$ is to be proven, it is enough to show that under the additional assumption we have a proof for D, we can find a proof for G:

$$\frac{\Gamma, X \in D \vdash_\Sigma P \in G}{\Gamma \vdash_\Sigma (\texttt{fun}\ X.P) \in D \to G}\ \mathsf{R}\!\to$$

To prove a universal quantified goal formula $\forall X : A_D.G$ in a context Γ, G has to be proven from an extended set of assumptions: Γ must be extended by the assumption that we have a proof term of the embedded data type A_D. Note, that we have to avoid to name the new assumption X, since X could already be defined in the context Γ. Instead, we name it Y, a new variable name — and then replace all occurrences of X in G by Y.

$$\frac{\Gamma, Y \in \overline{A_D} \vdash_\Sigma [Y/X]P \in [Y/X](G)}{\Gamma \vdash_\Sigma (\texttt{fun}\ X.P) \in \forall X : A_D.G}\ \mathsf{R}\forall$$

The existential rule on the right is an extended version of the one in the intuitionistic calculus. In the intuitionistic calculus, if a formula $\exists x.G$ is to be proven, it is enough to find a witness object a s.t. $[a/x](G)$ is provable. In the MLF setting, this idea remains the same: In addition, we have to make sure, that the witness term is a term of the right type.

The rule reads as follows: If $\exists X : A_G.G$ is to be proven from a context Γ, we have to make sure that there is a witness term P' which is of type $\overline{A_G}$ and $[P'/X](G)$ can be proven from Γ. The proof term we construct must now take both proof terms into account. We introduce a new program constructor (inx P' P) which represents the witness proof term and the proof term itself.

$$\frac{\Gamma \vdash_\Sigma P' \in \overline{A_G} \quad \Gamma \vdash_\Sigma P \in [P'/X](G)}{\Gamma \vdash_\Sigma (\texttt{inx}\ P'\ P)\ \in \exists X : A_G.G}\ \mathsf{R}\exists$$

So far we took only rules of the intuitionistic calculus and modified and extended them slightly. As we mentioned earlier, induction will play a major rôle in the proof system for MLF. The standard technique of tackling induction is to generate a set of induction principles for an inductively defined type [Hue88, PM93, C$^+$95]. The problem with induction principles is, that they are very rigid in their form. Proofs may not be easily found. We suspect that the inflexible character of induction principles my paralyze the proof process. We propose, to abandon the idea of proving well-foundedness within the system and to provide a rule which introduces the induction hypothesis in a more general way. This idea is realized in the recursion rule:

Recursion: The idea of the recursion rule is, that if a formula G is to be proven, we can simply assume it to use it as the induction hypothesis — which happen to have the same form as G. But now G must be transformed into a data formula. This is only possible, if we restrict recursion to core formulae C. The proof term we construct is a recursion operator. It captures the name of the induction variable — in our rule X, and the proof term. The recursive character of this program was described in section 3.2. The preliminary version of the rule is defined as follows:

$$\frac{\Gamma, X \in C \vdash_\Sigma P \in C}{\Gamma \vdash_\Sigma (\texttt{rec}\, X.P) \in C}\,\textsf{rec}$$

The definition rule is not complete yet. It has to be refined by a side condition. The following example shows that this formulation accepts derivation which should not count as valid derivations:

Example 3.27 *Let G be a goal which shall be proven from a context Γ. Omitting the side condition at the* rec*-rule we can easily establish the following derivation:*

$$\frac{\dfrac{}{\Gamma, X \in G \vdash_\Sigma X \in G}\,\textsf{id}}{\Gamma \vdash_\Sigma \texttt{rec}\, X.X \in G}\,\textsf{rec}$$

We cannot accept this derivation as a valid derivation. The intuition behind this derivation is: Assume G and prove G from this new assumption. X is the induction hypothesis. The problem lies within the non-totality of the proof term. A proof term witnesses the provability of a formula by itself, or it describes a concept of how to construct a witness program. Obviously these programs have to be total, that is a witness program has to be the result of an evaluation of the program. We make the notion of evaluation later more precise. The program $\texttt{rec}\ X.X$ *is not total: The application of* $((\texttt{rec}\, X.X)\ \overline{M})$ *yields after one reduction step* $((\texttt{rec}\, X.X)\ \overline{M})$. *The program will never terminate.*

To exclude non-total programs as proof terms, we introduce a side condition: $P \downarrow X$. The new judgement of the form $P \downarrow X$ holds if and only if the parameters to which X is applied are getting smaller according to a well-founded ordering. We will not go into the details how $P \downarrow X$ may be defined, we only set up the "interface" to the proof. The judgment $P \downarrow X$ is this interface. The recursion rule has the form:

$$\frac{\Gamma, X \in C \vdash_\Sigma P \in C}{\Gamma \vdash_\Sigma (\texttt{rec}\, X.P) \in C}\,\textsf{rec} \quad \text{with } P \downarrow X$$

Left Rules:

After introducing all right rules for MLF, we concentrate now on the left rules. The left rules are operating on the context. A left rule is applicable, if an assumption has a certain form. We will structure the presentation of the rules in two parts. In the first part we will give the rules which are taken almost directly from the proof system for intuitionistic logic. In the second part, we will present the rules, which bridge the gap between MLF and LF.

The definition of left rules will make use of the case construction for programs. The case distinction program is defined as

$$\left(\begin{array}{l} \texttt{case}\ P\ \texttt{of} \\ \quad Q^{(1)} \Rightarrow P^{(1)} \\ \quad \vdots \\ \mid \quad Q^{(n)} \Rightarrow P^{(n)} \end{array}\right)$$

The $Q^{(k)}$'s stand for patterns. The operational meaning of this program reads as follows: Assume that P is a program. P should be matched with $Q^{(k)}$, moreover there should be exactly one k s.t. $Q^{(k)}$ matches with P. The result of this matching process is a meta substitution: Θ. Under the eager evaluation ordering $\hookrightarrow_{MLF}$ the case construct reduces to $[\Theta](P^{(k)})$.

Before we define the left rules for MLF, the notion of patterns must be closer examined: Let the set $\{Q^{(1)}..Q^{(n)}\}$ define the set of patterns of the case distinction. This set of patterns should be sound on complete with respect to the different forms P can take. *Soundness* means, that there is at most one pattern $Q^{(k)}$ which matches with the canonical form of P. *Completeness* means, that there is at least one pattern $Q^{(k)}$ which matches with the canonical form of P. We make both notions more formal: A complete set of patterns is defined as:

Definition 3.28 (Complete set of patterns) *Let S be a set of patterns: S is called complete with respect to a goal formula G iff*

$$\Gamma \vdash_\Sigma P \in G \textit{ implies that a } Q \in S \textit{ matches with the canonical form of } P$$

and a sound set of patterns is defined as follows:

Definition 3.29 (Sound set of patterns) *Let S be a set of patterns: S is called sound with respect to a goal formula G iff*

$$Q \textit{ matches with } P \textit{ and } Q' \textit{ matches with } P \textit{ implies that } Q = Q'$$

As in the right case, there are some standard connectives, which have to be defined. These connectives are conjunction, disjunction, implication, universal and existential quantification. As before, we will examine rule by rule and comment on the changes and extensions in comparison with the proof system for intuitionistic logic.

Note, that we did not define any structural rules for this calculus. In section 4.2 we will see that that weakening and contraction are admissible rules in this system. It cannot be expected that a general exchange rule exists. Exchanging two assumption in a context may violate the dependencies of LF types from each other.

Since we do not allow any structural rules, we have to duplicate occurrences of the formula in question, from the conclusion to the premisses. We know that in a non-resource oriented logic, assumptions cannot disappear.

The first rule we discuss is the rule for conjunction. The rule is applicable in a bottom to top manner if a data formula $D_1 \wedge D_2$ can be found in the context. We then extend the context by two new assumptions, namely the assumption X_1 as a proof of D_1 and X_2 as a proof of D_2.

If we then can prove G, a proof term P will be provided. P may contain the two new variables X_1 and X_2. To construct a new proof term for the rule, we have to bring X, X_1, X_2 and P together: The proof term is a case construct, ranging over X. We already know that X is the proof of a conjunction and it is easy to show, that $\{(\texttt{pair}\ X_1\ X_2)\}$ is a sound and complete set of patterns for $D_1 \wedge D_2$. So, the final proof term has the form: $(\texttt{case}\ X\ \texttt{of}\ (\texttt{pair}\ X_1\ X_2) \Rightarrow P)$.

$$\frac{\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G}{\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2 \vdash_\Sigma (\texttt{case}\ X\ \texttt{of}\ (\texttt{pair}\ X_1\ X_2) \Rightarrow P) \in G}\ \mathsf{L}\wedge$$

Using the same idea, we define now the rule for disjunction. The rule reads as follows: If we have the assumption $D_1 \vee D_2$, and we want to prove a goal G, then we can use D_1 to prove G iff we can prove G also from D_2. Recall that we defined two constructors for proof terms of a disjunction: $\texttt{inl}, \texttt{inr}$. A sound and complete set of patterns is $\{(\texttt{inl}\ X_1), (\texttt{inr}\ X_2)\}$. The resulting proof term is a case distinction between these both constructors. The rule is formulated as follows:

$$\frac{\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, X_1 \in D_1 \vdash_\Sigma P_1 \in G \quad \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, X_2 \in D_2 \vdash_\Sigma P_2 \in G}{\Gamma_1, X \in D_1 \vee D_2, \Gamma_2 \vdash_\Sigma \begin{pmatrix} \texttt{case}\ X\ \texttt{of} & (\texttt{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\texttt{inr}\ X_2) \Rightarrow P_2 \end{pmatrix} \in G}\ \mathsf{L}\vee$$

The next rule is the rule for implication left. Here we assume, that we have an assumption of the form $G_1 \to D$. If we can prove that G_1 is true, i.e. that there is a proof term P_1 of G_1, we can use the function represented by X to obtain a proof term of G_2. If the original goal formula G_2 can now be proven from the an extended set of assumptions — extended by new assumption that there is a proof Y of D — then we are all set. Let P_2 be the proof term for the formula G_2. The proof term which is defined by the rule has to reflect the relationship between the Y and P_1. The intended meaning of the new program is: Instantiate the meta variable Y with the application of X to P. We have to combine two different programs to build up this proof term. First the program for instantiation is as follows: $\texttt{let}\ P_3\ \texttt{be}\ Y\ \texttt{in}\ P_2$. The program P_3 is derived by applying X to P_1: $(\texttt{app}\ X\ P_1)$. Here is the version of the rule implication on the left.

$$\frac{\Gamma_1, X \in G_1 \to D, \Gamma_2 \vdash_\Sigma P_1 \in G_1 \quad \Gamma_1, X \in G_1 \to D, \Gamma_2, Y \in D \vdash_\Sigma P_2 \in G_2}{\Gamma_1, X \in G_1 \to D, \Gamma_2 \vdash_\Sigma (\texttt{let}\ (\texttt{app}\ X\ P_1)\ \texttt{be}\ Y\ \texttt{in}\ P_2) \in G_2}\ \mathsf{L}\to$$

The next rule is universal quantification on the left. This rule is applicable if there is a universally quantified formula $\forall Y : A_G.D$ in the context. Let P_1 be a proof term of $\overline{A_G}$. This proof term can be interpreted as an LF object of type A_G. Since we can use the function X to obtain a proof term of type $[P_1/Y](D)$, the set of assumptions for can be extended by the assumption Z is a proof term of formula $[P_1/Y](D)$. If G is now provable, we obtain a proof term P_2. The proof of the rule is constructed in the same way as in rule $\mathsf{L}\to$: application and instantiation have to be combined. The proof term has the form: $(\texttt{let}\ (\texttt{app}\ X\ P_1)\ \texttt{be}\ Z\ \texttt{in}\ P_2)$.

$$\frac{\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma P_1 \in \overline{A_G} \quad \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, Z \in [P_1/Y](D) \vdash_\Sigma P_2 \in G}{\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma (\texttt{let}\ (\texttt{app}\ X\ P_1)\ \texttt{be}\ Z\ \texttt{in}\ P_2) \in G}\ \mathsf{L}\forall$$

The existential rule on the left corresponds to the conjunction rule on the left. It is applicable if $\exists Y : A_G.D$ is an assumption in the context. This implies, that the witness object is available and also a proof term of $[X_1/Y](D)$, where X_1 represents the witness object. This proof term is represented by X_2. Let P be a proof term of the goal formula G proven from this extend set of assumptions. It can be shown that $\{(\texttt{inx}\ X_1\ X_2)\}$ is a sound and complete set of patterns for $\exists Y : A_G.D$. Therefore the proof term has the form : $(\texttt{case}\ X\ \texttt{of}\ (\texttt{inx}\ X_1\ X_2) \Rightarrow P)$.

$$\frac{\Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D) \vdash_\Sigma P \in G}{\Gamma_1, X \in \exists Y : A_D.D, \Gamma_2 \vdash_\Sigma (\texttt{case}\ X\ \texttt{of}\ (\texttt{inx}\ X_1\ X_2) \Rightarrow P) \in G}\ \mathsf{L}\exists$$

This concludes the presentation of the set of rules which are defined a long the lines of [Gal93]. Next, we define a set of rules, which treats embedded Π-abstraction as data formulae.

LF related rules Assume we have an assumption of a functional LF type in the actual context. This assumption is of the form: $\overline{\Pi x : A_G.A_D}$. MLF is allowed to look inside the embedding function. Assume we can find a proof term of the formula $\overline{A_G}$. This proof term might be of arbitrary form P'. Since we know that P' can be projected to the LF level, we can define an additional assumption $\overline{((\Pi x : A_G.A_D)\ \underline{P'})}$ with which G is to be proven. It is obvious that the embedded type is not pure. To preserve purity, we restrict the form of the proof term to be either $\overline{M}$ or a metavariable X. We obtain the following rules:

$$\frac{\begin{array}{l}\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2 \vdash_\Sigma \overline{M} \in \overline{A_G} \\ \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, Y \in \overline{\{M/x\}_{\text{type}}(A_D)} \vdash_\Sigma P \in G\end{array}}{\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2 \vdash_\Sigma [\underline{X}\ \overline{M}/Y](P) \in G}\ \mathsf{L\Pi}$$

and

$$\frac{\begin{array}{l}\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2 \vdash_\Sigma Z \in \overline{A_G} \\ \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, Y \in \overline{\{\underline{Z}/x\}_{\text{type}}(A_D)} \vdash_\Sigma P \in G\end{array}}{\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2 \vdash_\Sigma [\underline{X}\ \underline{Z}/Y](P) \in G}\ \mathsf{L\Pi V}$$

We will discuss the notion of purity in MLF in more detail in section 4.1. It seems as if we restricted the application possibilities for this rule quite a lot. But it turns out that in all the examples we were experimenting with, no negative indication occured that restriction to $\overline{M}$ and X is to strong.

The second LF related rules are as follows: Both previous rules assumed an embedded Π-type to be declared in the context. This is not the only place where Π-types are defined: constants of Π-types are defined in the signature. The motivation for the next rule is to connect MLF to the signature. The rules are very similar to $\mathsf{L\Pi}$ and $\mathsf{L\Pi V}$ — the only difference is, that $\Pi x : A_G.A_D$ is part of the signature and not of the context. Note, that we have to apply the same argument to motivate the restriction of considering only programs $\overline{M}$ and meta variables Z and not arbitrary programs of the form P' as proof terms of $\overline{A_G}$. We define the following two rules:

$$\frac{\Gamma \vdash_\Sigma \overline{M} \in \overline{A_G} \quad \Gamma, Y \in \overline{\{M/x\}_{\text{type}}(A_D)} \vdash_\Sigma P \in G}{\Gamma \vdash_\Sigma [\overline{c\,M}/Y](P) \in G} \text{L}\Pi\Sigma \quad \text{where } c : \Pi x : A_G.A_D \in \Sigma$$

and

$$\frac{\Gamma \vdash_\Sigma Z \in \overline{A_G} \quad \Gamma, Y \in \overline{\{\underline{Z}/x\}_{\text{type}}(A_D)} \vdash_\Sigma P \in G}{\Gamma \vdash_\Sigma [\overline{c\,\underline{Z}}/Y](P) \in G} \text{L}\Pi\Sigma\text{V} \quad \text{where } c : \Pi x : A_G.A_D \in \Sigma$$

To complete the inference rule system of MLF we have to define two more rules. The case distinction rule is still missing and also the cut rule, which allows application, and the reuse of already proven lemmata.

Case Distinction

Next, we introduce the case distinction rule. For inductive proofs over inductively defined types case distinction is a common proof strategy. Several systems which can perform meta level reasoning, like Coq [C+95], PVS [ORS92, RSC95] and others, possess an inductive proof component. They are based on the generation of induction principles. MLF does not follow this idea but provides recursion and case distinction as the major components for inductive proofs. We outline the motivation behind the definition of the case distinction rule.

Assume we want to derive a judgment of the form $\Gamma \vdash_\Sigma Q \in G$ where X is a free variable in G, and possibly also in Γ. X must be defined in Γ, otherwise Γ is not well-formed. Suppose that X is a variable of the goal formula $\overline{A_G}$. The idea is, that if we find a proof term P of $\overline{A_G}$ we would like to restrict the proof by considering the different forms the program P can take. The form of P leads to a set of subproofs, where Γ and G are refined.

The form of the program P is arbitrary. Using an appropriate reduction relation, P can be rewritten to $\overline{M}$. M is an LF object of type A_G in context Γ. The different forms M can take is defined by the signature. Since Σ is fixed, it is possible to derive a sound and complete set of patterns for A_G. But how can this set be determined? The question is challenging because A_G is a dependent type.

A sound and complete set of patterns for A_G can be found by selecting a certain subset of the signature Σ — call it Σ' for now. Σ' should contain only object constant declarations and not type constant declarations: For all object constant declarations $c : A_D$ in Σ' the following must hold: A_G must be refinable into a type A_D — but A_D might be not atomic. In general it has the form

$$A_D = \Pi x_1 : A_{G1}..\Pi x_n : A_{Gn}.\ A_P$$

If we assume that there are proof terms $X_1 \in \overline{A_{G1}}$ up to $X_n \in \overline{A_{Gn}}$, we can transform A_D into an atomic type:

$$A_P{}' = \{\underline{X_1}/x_1..\underline{X_n}/x_n\}(A_P)$$

If this type $A_P{}'$ is a refinement of the original type A_G, then $(c\,\underline{X_1}..\underline{X_n})$ is a pattern of an object of the refined type A_G.

We will now make this notion of refinement more precise. Σ' was defined as a subset of the signature. Every declaration in Σ' contributes to a refinement of the type A_G. The refinement is essentially a substitution which unifies A_G and A_P'. These observations lead to the definition of the refinement of a type:

Definition 3.30 (Refinement of types) $\langle \Lambda, \Theta \rangle$ *is a refinement of* $X \in A_G$ *in context* Γ *using* $c : \Pi x_1 : A_{G1} .. \Pi x_n : A_{Gn}.\ A_P$ *if and only if the following conditions hold:*

1. $\{X_1..X_n\}$ *are fresh meta variables in* Γ
2. $\Theta' = \text{unify}(\{\underline{X_1}/x_1..\underline{X_n}/x_n\}(A_P) \approx A_G)$
3. $\Theta = \Theta'|_{dom(\Gamma)} \cup (\overline{c\ \underline{X_1}..\underline{X_n}}/X)$
4. Λ *is inductively defined as* Λ_n*:*

$$\begin{array}{rcl} \Lambda_0 & = & \cdot \\ \Lambda_{i+1} & = & \Lambda_n, X_{i+1} \in \overline{\{\underline{X_1}/x_1..\underline{X_i}/x_i\}_{type}(A_{Gi+1})} \end{array}$$

Note, that the unification problem is a real unification problem and not only a matching problem. This becomes immediately evident, when looking at the following example: Assume that we can derive a program P from a context Γ of type $F \in \overline{\text{eval}\ \underline{Y}\ (\text{s z})}$. Obviously, the meta-variable Y must be declared in the context Γ. If we perform the unification algorithm to match this formula with $\overline{\text{eval}\ (\text{s}\ \underline{X_1})\ (\text{s}\ \underline{X_2})}$ we obtain the following substitution Θ:

$$\Theta = \overline{(\text{s}\ \underline{X_1})}/Y, \overline{\text{z}}/X_2$$

Strictly speaking, the substitution consists of two parts, a refinement part and a so called verification part: The *refinement substitution* assigns terms made of new variables to old variables. In this example, the refinement substitution has the form $\Theta_r = [\overline{(\text{s}\ \underline{X_1})}/Y]$. The other part of the substitution we is called *verification* substitution. In this case the newly introduced variables are assigned to subterms. In the example above, the verification substitution has the form: $\Theta_v = [\overline{\text{z}}/X_2]$. Obviously, z is not further refinable. The verification part of a substitution therefore does not contribute to the form of the refinement, but it simply carries the information that it is possible to make the given term and the term taken from the signature equal. The verification part of the substitution is not of our concern, it is handled by the rule as we will see, when we actually define the **case** rule. The part of the substitution we need is the refinement substitution. To obtain the refinement substitution we have to restrict the substitution Θ in the definition of $Ind_{\Sigma,\Gamma}(X, B)$ to the domain of Γ.

Based on this definition, we define a set of all possible refinements: $\text{Ind}_{\Sigma,\Gamma}(X, A_G)$: X is the variable, induction is made over, $\overline{A_G}$ its formula. To obtain now the set of all possible constructors, we have to select all those declaration $c : A_D$ which may refine A_G. $\text{Ind}_{\Sigma,\Gamma}(X, A_G)$ is a set of pairs $\langle \Lambda, \Theta \rangle$:

Definition 3.31 (Inductive type) *Let* Γ *be context,* Σ *be a signature,* X *a variable, and* A_G *be an atomic LF type. The inductive type of* X, A_G *with respect to* Γ *and* Σ *is defined as*

$$Ind_{\Sigma,\Gamma}(X, A_G) =$$
$$\{\langle \Lambda, \Theta \rangle | \langle \Lambda, \Theta \rangle \textit{ is a refinement of } X \in A_G \textit{ in context } \Gamma \textit{ using } c : A \textit{ in } \Sigma\}$$

In the definition of the **case** rule we distinguish between all different refinements in $\mathrm{Ind}_{\Sigma,\Gamma}(X, A_G)$. The **case** rule has therefore as many premisses as $Ind_{\Sigma,\Gamma}(X, A_G)$ has elements.

There are many different ways of how to define the **case** rule. For our system, we want the **case** rule to have the following property: The inductive argument is assumed to be carried out for a more general atomic type. When specializing the atomic type by a substitution η, the same proofs of the premisses should be still valid. η incorporates the verification part of the substitution Θ_v. Here is the rule:

$$\frac{\vdash_\Sigma [P/X](\Gamma)\ \mathrm{ctx} \quad \Gamma \vdash_\Sigma P \in \overline{A_G} \quad \overset{\text{for all } i \leq n}{\Lambda^{(i)}, [\Theta^{(i)}](\Gamma') \vdash_\Sigma P^{(i)} \in [\Theta^{(i)}](G')}}{[P/X](\Gamma) \vdash_\Sigma \left(\begin{array}{l} \textsf{case } P \textsf{ of} \\ \quad c_1\ \overline{Y_1^{(1)}...Y_{m_1}^{(1)}} \Rightarrow [\eta](P^{(1)}) \\ \quad ... \\ \mid \quad c_n\ \overline{Y_1^{(n)}...Y_{m_n}^{(n)}} \Rightarrow [\eta](P^{(n)}) \end{array} \right) \in [P/X](G)}\ \textsf{case}$$

where following side conditions hold:

1. $\mathrm{Ind}_{\Sigma,\Gamma}(X, A_G') = \{\langle\Lambda^{(1)}, \Theta^{(1)}\rangle..\langle\Lambda^{(n)}, \Theta^{(n)}\rangle\}$
2. There is a η s.t. $[\eta](A_G') = A_G$, $[\eta](\Gamma') = \Gamma$ and $[\eta](G') = G$

Lemma 3.32 *$\Lambda^{(i)}$ and $\Theta^{(i)}$ enjoys the following properties:*

$$Free(\Lambda^{(i)}) \subset \{D_1^{(i)}, ..., D_{m_i}^{(i)}\} \tag{3.6}$$
$$Var(\Lambda^{(i)}) = \{D_1^{(i)}, ..., D_{m_i}^{(i)}\} \tag{3.7}$$

$$\vdash_\Sigma \Lambda^{(i)}\ ctx \tag{3.8}$$
$$\Theta^{(i)}\ \textit{is strictly pure} \tag{3.9}$$

Proof: For all $\langle\Lambda^{(i)}, \Theta^{(i)}\rangle \in Ind_{\Sigma,\Gamma}(X, A_G)$, the A_D's in the definition $\langle\Lambda^{(i)}, \Theta^{(i)}\rangle$ are declared in the signature and therefore closed. Consequently (3.6) and (3.7) hold. $\Theta^{(i)}$ is strict because of construction, an inductive argument yields $\vdash_\Sigma \Lambda^{(i)}$ ctx □

The Cut rule

The definition of the cut rule is similar to the one introduced in [Pfe94c]. The definition is not straightforward because of dependencies: the problem lies in the choice of how to relate the contexts of the premisses with the context of the conclusion. For every element $X \in D$ in the context Γ of a sequent, all introduced variables which are introduced in Γ before X can possibly occur in D.

Looking at the cut rule from top to bottom, the cut rule can be interpreted as a way to perform application: Assume we have a proof of $\Gamma \vdash_\Sigma (\texttt{fun}\ X.P) \in \forall X : A_D.G$. This sequent can be read as: In a certain context Γ, a program was found which for every element in A_D yields a proof term of goal formula G. Suppose now that we have a different sequent, $\Gamma_1, F \in \forall X : A_G.G, \Gamma_2 \vdash_\Sigma P' \in G'$. This sequent reads as follows: Under the assumption that we have a program of the data formula $\forall X : A_G.G$, we effectively can construct a proof term of some goal formula G'. The purpose of the cut rule is it to combine both proof terms to a proof term which witnesses the following judgment: From the remaining contexts of both participating sequents a proof term for the goal formula G' can be constructed.

Because of the definition of goal and data formula, the universally quantified formulae might not be identical in both sequents. We therefore must restrict the cut formula to be a core formula — as in the case for recursion. We have seen that every core formula is a data formula and a goal formula.

The cut rule is defined as follows: Two sequents can be cut with each other, if a core formula C occurs as a goal formula in one sequent and as a declaration $X \in C$ in the context of the other sequent. The declaration of the cut formula from the second sequent is removed and every free occurrence of X is replaced by the proof term of the first. Note, that the variable X cannot occur free in the context of the first sequent. The context of the first sequent, must be the initial context of the second. In the second context the meta variable X is already declared. Therefore the second sequent wouldn't be provable because the context is not well-formed. Here is the definition of the cut-rule:

$$\frac{\Gamma_1 \vdash_\Sigma P \in C \quad \Gamma_1, X \in C, \Gamma_2 \vdash_\Sigma P' \in G}{\Gamma_1, [P/X](\Gamma_2) \vdash_\Sigma [P/X](P') \in [P/X](G)}\,\textsf{cut}$$

This concludes the set of typing rules for $\Gamma \vdash_\Sigma P \in G$. In the next chapter we will describe some theoretical properties of MLF.

Chapter 4

Theoretical Aspects of MLF

This chapter states some proof theoretic results about MLF. The system as we have defined has to be analyzed appropriately. In section 3.3 we defined the difference between objects and pure objects. We define now two differently strong notions of purity preservation. Then we show that MLF, as we defined it preserves purity according to the weaker notion. If we refine the system slightly, we can prove purity preservation in the stronger sense.

The second result we show is a local reduction theorem. The local reduction theorem is the first step towards a cut elimination theorem. A general cut elimination theorem might be to general — if provable at all — and its proof would be beyond the scope of this thesis. Important results on the way to the local reduction theorem are the admissibility of weakening and contraction.

This chapter is organized as follows: In the first section we give the purity results, and the second we give the local reduction proof.

4.1 Purity results

The notion of purity was defined to distinguish between objects which can depend on meta variables alone and objects which depend on composite programs. In this section we examine how purity and MLF go together. We show, that the system we defined in section 3.3 preserves purity in the way, that if all contexts, programs and formula participating in the premisses are pure, then the context, program and formula in the conclusion are pure. In a next step we will generalize the inference system of MLF to a stronger inference system, called the inference system for pure MLF. Now we can prove for the cut free fragment, that if $\Gamma \vdash_{\Sigma} P \in G$ is derivable, and Γ and G are pure, then the resulting program P must be pure, too. We call both results purity preservation results. In the first case, purity is preserved with respect to rule application, in the second, purity is preserved with respect to derivations.

4.1.1 Basic Results

In section 3.1 we also introduced the notion of pure and strictly pure substitutions. A pure substitution is a substitution in which are participating programs are pure. It is obvious, that the application of a pure substitution to a pure object may generate an impure object. Since we want to say something about pure objects, it makes sense to strengthen the definition of pure substitution, to classify those, which generate pure programs. We call those substitutions strictly pure. It is also clear, that the participating programs must not longer be arbitrary programs but meta variables and embedded LF objects.

For strictly pure substitutions, we have to show some basic results first: Strictly pure substitutions are doing what they are supposed to do, namely to preserve purity when being applied to something. Second, we show that the reverse also holds: If we have for a example an object which is the result of the application of an arbitrary substitution to it, and it is known that this result object is pure, then the original object must have been pure. Or, to put it the other way around: No impure object can turn pure by applying a substitution to it. A third result is similar to the second. If we have a pure object, which is the result of a substitution on an object M, and this result is pure, then the substitution must be strict, at least in the variables, which are used to produce the result object. These variables are the free variables occuring in M. Or again, we can put this result the other way around. If we substitute into an object programs which are neither meta variables, nor the embedded LF objects, then the result cannot be pure

We will give these basic results first for objects and types, then for programs, formulae and finally for contexts. Since we have dependencies between objects and types, we have to prove some of the theorems by mutual induction. Here are the three theorems for pure objects and pure types. The proofs are done by induction, the detailed proofs are given in appendix B.

Basic Results for Objects and Types

By mutual induction we can show, that strict Θ applied to objects or types preserve purity.

Lemma 4.1 *Let M' be a pure object, A' a pure type, and Θ a strictly pure substitution, i.e. $\Theta(X) = Y$ or $\Theta(X) = \overline{M}$ for all $X \in dom(\Theta)$. Then $\Theta(M')$ is pure and $\Theta(A')$ is pure.*

On the other side, we can show that substitutions cannot purify objects

Lemma 4.2 *Let Θ be a substitution and $[\Theta](M)$ be a pure object. Then M is pure.*

and types

Lemma 4.3 *Let Θ be a substitution and $[\Theta](A)$ be a pure type. Then A is pure.*

The third result has to be shown by mutual induction on objects and types. If Θ generates a pure result object/type, then Θ must be strict at least in the variables, which are used to produce the result object:

Lemma 4.4 *, Let Θ be substitution, M object and $[\Theta]_{object}(M)$ pure and A be a type and $[\Theta]_{type}(A)$ pure. Then $\Theta|_{Free(M)}$ must be strict and $\Theta|_{Free(A)}$ is strict.*

Basic Results for Programs

Strict Θ are preserving purity for programs:

Lemma 4.5 *Let P' be a pure program and Θ a strictly pure substitution. Then $\Theta(P')$ is pure.*

And impure programs cannot be purified by substitution application:

Lemma 4.6 *Let Θ be a substitution and $[\Theta](P)$ be a pure. Then P is pure.*

The third result we had for objects and types, will surely not be available for programs: Assume Θ be a substitution, and $P = (\texttt{pair}\ D_1\ D_2)$ a program, with D_1, D_2 variables. $\Theta(P) = (\texttt{pair}\ \Theta(D_1)\ \Theta(;D_2))$ is pure program, even for arbitrary pure programs $\Theta(D_1)$ and $\Theta(D_2)$.

Basic Results for Formulae

The results in the case of formulae are straightforward. All the lemmas are easy to prove by induction and we obtain the following three lemmata:

Lemma 4.7 *Let G' be a pure formula and Θ a strictly pure substitution. Then $\Theta(G')$ is pure.*

Lemma 4.8 *Let Θ be a substitution and $[\Theta](G)$ be a pure formula. Then G is pure.*

Lemma 4.9 *Let Θ be substitution, G formula and $[\Theta]_{formula}(G)$ pure. Then $\Theta|_{Free(G)}$ is strict.*

It is clear, that the proofs of these lemmata will refer to the basic results we obtained for types since types may be embedded into formulae.

Basic Results for Contexts

And finally, we prove two of the basic results for contexts. We will not need a strictness lemma for the purity analysis of MLF.

Lemma 4.10 *Let Γ' be a pure context and Θ a strictly pure substitution. Then $\Theta(\Gamma')$ is pure.*

Lemma 4.11 *Let Θ be a substitution and $[\Theta](\Gamma)$ be a pure context. Then Γ is pure.*

So far, we presented some properties of objects, types, programs and formulas. Strictly pure substitutions preserve purity. We now want to tackle a bigger problem: What can be said about the type system for MLF rules? We consider two properties of MLF:

1. Purity Preservation of typing rules: If every context, program and goal formula are pure in the premiss of a rule, then the context, program, and goal are pure in the conclusion.
2. Purity preservation of derivations: If we have a proof of $\Gamma \vdash_\Sigma P \in G$ and we assume Γ, G pure, then P is pure.

To enforce the second question, we have to change the set of rules. For example the rule L$\forall$ has the premiss $\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma P_1 \in \overline{A_G}$. Under the assumption that the context $\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2$ is pure and therefore $\overline{A_G}$ pure we could prove that the proof term P is pure. But, when looking at the second premiss: $\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, Z \in [P_1/Y](D) \vdash_\Sigma P_2 \in G$ we see, that the substitution P_1/Y is not strictly pure. As mentioned earlier purity is not necessarily preserved by a non-strict substitution application. There is one remedy to this: The proof system has to be weakened by enforcing the substitution to be pure. We will show this in subsection 4.1.3. In the next subsection we address the first property.

4.1.2 Purity Preservation of MLF Rules

We will now prove the weaker formulation of the purity result for the MLF inference system. We remark, that we have to establish a side condition for the case rule for this argument to go through. The side condition is as follows: In the premiss of the rule, we have the formula $\overline{A_G}$, which defines implicitly the different cases: A_G is an instantiation of $A_G{}'$, which is used to define $\mathrm{Ind}_{\Sigma,\Gamma}(X, A_G{}')$. In the original formulation of the rule nothing is said about the purity of $A_G{}'$. We have to assume that $A_G{}'$ is pure. The refined version of the rule is given below.

$$\frac{\vdash_\Sigma [P/X](\Gamma)\ \mathrm{ctx} \quad \Gamma \vdash_\Sigma P \in \overline{A_G} \quad \overset{\text{for all } i \le n}{\Lambda^{(i)}, [\Theta^{(i)}](\Gamma') \vdash_\Sigma P^{(i)} \in [\Theta^{(i)}](G')}}{[P/X](\Gamma) \vdash_\Sigma \left(\begin{array}{l} \mathsf{case}\ P\ \mathsf{of} \\ \quad \overline{c_1\ \underline{Y_1^{(1)}} ... \underline{Y_{m_1}^{(1)}}} \Rightarrow [\eta](P^{(1)}) \\ \quad ... \\ \mid \quad \overline{c_n\ \underline{Y_1^{(n)}} ... \underline{Y_{m_n}^{(n)}}} \Rightarrow [\eta](P^{(n)}) \end{array} \right) \in [P/X](G)}\ \mathsf{case}'$$

where following side conditions hold:

1. $A_G{}'$ is pure
2. $\mathrm{Ind}_{\Sigma,\Gamma}(X, A_G{}') = \{\langle\Lambda^{(1)}, \Theta^{(1)}\rangle .. \langle\Lambda^{(n)}, \Theta^{(n)}\rangle\}$
3. There is a η s.t. $[\eta](A_G{}') = A_G$, $[\eta](\Gamma') = \Gamma$ and $[\eta](G') = G$

It seems to us, as if this additional restriction does not restrict the applicability for the rule at all. The framework is designed to reason about LF. Therefore we expect the type AG' to be an LF type about which induction should be made — a type which is a generalization of AG: there exists a η, s.t $A_G = [\eta](A_G{}')$. Assume, that AG' is impure. We can easily construct a more general LF type AG'', which result from AG' by replacing all impure subprograms by fresh variables. This can be expressed as: there is a substitution η', and a pure $A_G{}''$, s.t. $A_G{}' = [\eta'](A_G{}'')$. Hence $A_G{}''$ is a pure generalization of A_G: $A_G = [\eta \circ \eta'](A_G{}'')$.

We conjecture that the case rule in the MLF framework can be replaced by case', without loss of generality.

Theorem 4.12 *Every typing rule in MLF without cut is purity preserving. That is, when the premisses are pure (all participating objects, types, programs, formulae, and contexts are pure) the conclusion will be pure, too.*

Proof: Case: case$'$: We can assume Γ to be pure, therefore $[P/X]$ is strictly pure and by lemma 4.10 we obtain $[P/X](\Gamma)$ is pure. From lemma 4.7 we obtain that $[P/X](G)$ is pure. By assumption $\overline{A_G}$ is pure, therefore A_G is a pure type. $[\eta](A_G{}')$ is pure by the side condition of the rule, therefore $\eta|_{Free(A_G{}')}$ is strictly pure by lemma 4.4. Since $\eta = \eta|_{Free(A_G{}')}$, η is a strictly pure substitution. Consequently for all i: $[\eta](P^{(i)})$ is pure, and therefore the proof term is a pure program.

Other cases: straightforward.

□

Theorem 4.12 shows, that the rules of MLF preserve purity. This result is very useful, because the framework we designed should be restricted to pure objects, pure types, pure formulae and pure programs only. Committing ourselves to consider only a system of inference rules which preserves purity in this sense, does not seem to be of any disadvantage in terms of expressive power. In contrary, it makes it very clear, which LF object and which LF type is used where and how. In the following subsection we will even go a step further and restrict the inference system for MLF a little more. The result is a system we call pure MLF.

4.1.3 Pure MLF

The goal of this subsection is to refine the inference system of MLF to obtain a system we call pure MLF. Pure MLF has the advantage, that form a derivation of $\Gamma \vdash_\Sigma P \in G$ and Γ, G pure in cut free MLF, P can be shown to be pure.

We demonstrate the underlying idea at the example of the existential rule on the right: It has has currently the following form:

$$\frac{\Gamma \vdash_\Sigma P' \in \overline{A_G} \quad \Gamma \vdash_\Sigma P \in [P'/X](G)}{\Gamma \vdash_\Sigma (\mathbf{inx}\, P'\ P) \ \in \exists X : A_G.G}\,\mathsf{R}\exists$$

We want to avoid the generation of any impure objects, types, programs or formulae: therefore the rules must be restricted to work on guaranteed pure version of objects, types, programs and formulae. We observe that the rule $\mathsf{R}\exists$ does not fulfill this criteria, because $[P'/X](G)$ may not be pure. This can happen because nothing is said about the form of P'. Since $[P'/X](G)$ must be a well formed formula, we know because of lemma 4.9 that the substitution $[P'/X]$ must be strict. Consequently P' can have two forms: $P' = Y$ or $P' = \overline{M}$. The existential rule can therefore be refined into the following two rules:

$$\frac{\Gamma \vdash_\Sigma Y \in \overline{A_G} \quad \Gamma \vdash_\Sigma P \in [Y/X](G)}{\Gamma \vdash_\Sigma (\mathbf{inx}\, Y\ P) \ \in \exists X : A_G.G}\,\mathsf{R}\exists'$$

and the rule where P' is replaced by $\overline{M}$:

$$\frac{\Gamma \vdash_\Sigma \overline{M} \in \overline{A_G} \quad \Gamma \vdash_\Sigma P \in [\overline{M}/X](G)}{\Gamma \vdash_\Sigma (\mathbf{inx}\, \overline{M}\ P) \ \in \exists X : A_G.G}\,\mathsf{R}\exists''$$

The same argument can be applied to L$\forall$ and we obtain the two refined version of L$\forall$:

$$\frac{\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma X_1 \in \overline{A_G} \quad \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, Z \in [X_1/Y](D) \vdash_\Sigma P_2 \in G}{\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma (\texttt{let}\ (\texttt{app}\ X\ X_1)\ \texttt{be}\ Z\ \texttt{in}\ P_2) \in G}\ \mathsf{L}\forall'$$

and

$$\frac{\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma \overline{M} \in \overline{A_G} \quad \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, Z \in [\overline{M}/Y](D) \vdash_\Sigma P_2 \in G}{\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma (\texttt{let}\ (\texttt{app}\ X\ \overline{M})\ \texttt{be}\ Z\ \texttt{in}\ P_2) \in G}\ \mathsf{L}\forall''$$

A third refinement we have to do is again concerned with the **case** rule. In the previous subsection we refined **case**, by establishing a new side condition: $A_G{}'$ has to be pure. This will not be enough for our following considerations: We have to establish a second side condition: A_G by itself must be pure, too. Here is the new rule **case**$''$:

$$\frac{\vdash_\Sigma [P/X](\Gamma)\ \text{ctx} \quad \Gamma \vdash_\Sigma P \in \overline{A_G} \quad \overset{\text{for all } i \leq n}{\Lambda^{(i)}, \Theta^{(i)}(\Gamma') \vdash_\Sigma P^{(i)} \in \Theta^{(i)}(G')}}{[P/X](\Gamma) \vdash_\Sigma \left(\begin{array}{ll} \multicolumn{2}{l}{\texttt{case}\ P\ \texttt{of}} \\ & \overline{c_1\ \underline{Y_1^{(1)}}\ldots\underline{Y_{m_1}^{(1)}}} \Rightarrow [\eta](P^{(1)}) \\ & \ldots \\ | & \overline{c_n\ \underline{Y_1^{(n)}}\ldots\underline{Y_{m_n}^{(n)}}} \Rightarrow [\eta](P^{(n)}) \end{array} \right) \in [P/X](G)}\ \mathsf{case}''$$

where following side conditions hold:

1. A_G, $A_G{}'$ is pure
2. $\text{Ind}_{\Sigma,\Gamma}(X, A_G{}') = \{\langle \Lambda^{(1)}, \Theta^{(1)} \rangle .. \langle \Lambda^{(n)}, \Theta^{(n)} \rangle\}$
3. There is a η s.t. $[\eta](A_G{}') = A_G$, $[\eta](\Gamma') = \Gamma$ and $[\eta](G') = G$

We call this refined version of the inference system for MLF as *pure MLF*. We have mentioned at the beginning of this chapter, that the inference system of MLF is not powerful enough to prove the following observation: Whenever Γ is a pure context, and G is a pure formula and a derivation $\Gamma \vdash P \in G$ holds, then P is a pure program. In pure MLF, we have all the ingredients to make the proof go through:

Theorem 4.13 (Generalized Purity Preservation) *Let Γ be a pure context, G a pure formula, P a program, and $\mathcal{D}$ a derivation of $\mathcal{D} :: \Gamma \vdash_\Sigma P \in G$ in the inference system of pure MLF without Cut. Then P is pure.*

Proof: By induction over the derivation $\mathcal{D}$:

Case: case$'$: $[P/X](\Gamma)$ is pure. If $X \in \mathit{Free}(\Gamma)$ then Γ is pure by lemma 4.6, else Γ is pure. If $X \in \mathit{Free}(G)$ then G is pure by lemma 4.8, else G is pure. B is pure and B' is pure, and therefore $[\eta](B')$ is pure. By lemma 4.4 we obtain, that $\eta = \eta|_{\mathit{Free}B'}$ is strict. We have $[\eta](\Gamma')$ is pure (because Γ is pure) and because of lemma 4.11 we obtain Γ' is pure. The same argument holds for $[\eta](G')$, which is pure since G is pure. Because of lemma 4.8 we know that G' is pure, too. Because of construction $\Theta^{(i)}$ is pure for all i. Thus, $\Theta^{(i)}(\Gamma')$ is pure. $\Lambda^{(i)}$ is pure, too, because of construction. And finally $\Theta^{(i)}(G')$ is pure because G is pure. Therefore we can apply the induction hypothesis and obtain that the $P^{(i)}$'s are pure for all i. Therefore the $[\eta](P^{(i)})$ are pure, and hence the proof term

$$\left(\begin{array}{l}\mathsf{case}\ P\ \mathsf{of}\\ \quad c_1\ \overline{\underline{D_1^{(1)}}\ldots\underline{D_{m_1}^{(1)}}} \Rightarrow [\eta](P^{(1)})\\ \quad \ldots\\ \mid\ \ c_n\ \overline{\underline{D_1^{(n)}}\ldots\underline{D_{m_n}^{(n)}}} \Rightarrow [\eta](P^{(n)})\end{array}\right)$$

is a pure program.

Case: All other cases straightforward.

□

4.1.4 Result

In this section, we analyzed the notion of purity with respect to MLF and a refined version of MLF. We saw, that both systems, MLF and pure MLF preserve purity. Working in pure MLF has the big advantage, that if we do not use the Cut rule in proofs, a purely stated goal G implies that the program P is pure. This is because theorems are stated as $\cdot \vdash_\Sigma P \in G$ with an empty context. Therefore, if the goal G is pure and since **cut** is not used, then the program we obtain will be pure. Furthermore, theorem 4.13 is a little more general then that: It also shows, that the results of every subderivation in pure MLF is pure, that is context, programs, and goal formulae will all be pure. We can therefore conclude, that for proof search without cut, the underlying object theory can be restricted to pure objects only.

4.2 Local reductions

In this section we show the local reduction property of MLF. Local reductions are part of the cut elimination proof. At the moment it is not clear, if a general cut-elimination result holds at all or not. This section is divided into two parts. In the first part we give some lemmata which are necessary to perform the proof of local reductions theorem. The proofs of the lemmata can be found in the appendix C. In the second part of this section we discuss the problem of local reductions.

4.2.1 Admissibility of Weakening and Contraction

As we have seen in the definition of the MLF inference rule system, there are no structural rules defined. The reason was, that we defined the system along the lines of LJ, as described in [Pfe94c]. The motivation to forget about structural rules and to show their admissibility later, is easily motivated. First of all we want to reduce the complexity of the inference rule system. Second — as shown in [Pfe94c] — cut elimination as a proof becomes suddenly feasible by structural induction and not any more by induction over a complexity measure.

The intuitionistic sequent calculus contains three structural rules. Weakening allows to add additional formulae to the context, that is the set of formulae on the lefthandside of the sequent symbol. Then there is contraction, which allows the removal of identical copies of formulae from the context. And finally there is the exchange rule. The exchange rule makes context independent of the order of the assumptions. We cannot expect the admissibility of the exchange rule for MLF, since formulae can dependent on variables which have to be introduced earlier into the context, exchange would destroy this property. But fortunately, we can show the admissibility of weakening and contraction:

Let $\mathcal{D} :: \Gamma \vdash_\Sigma P \in G$ be a derivation of a sequent. Weakening $\mathcal{D}$ means to add a new assumption into the context Γ. The position of the inserted assumption is essential because the order of the assumptions reflect the dependencies of the types. To insert an assumption $X \in G$ into Γ we have to split Γ into Γ_1 and Γ_2. To express that $\mathcal{D}$ is a derivation which is weakened by inserting $X \in G$ after Γ_1 we write $\mathcal{D}[\Gamma_1 \bigvee X \in G]$. First a little preparatory lemma:

Lemma 4.14 (Context extension) *Let Γ_1, Γ_2 be contexts, s.t. $\vdash_\Sigma \Gamma_1, \Gamma_2$ ctx. Let D be formula, s.t. $\Gamma_1 \vdash_\Sigma D$ data, then $\vdash_\Sigma \Gamma_1, X \in D, \Gamma_2$ ctx*

Proof: see appendix C □

Recall, that we do not check the context in every rule. This is done only in the leaves. While applying rules, the context shrinks. It is easily seen, that this shrinking process cannot invalidate the context — except in the case rule. But here, we make sure, that the context is really a context, by adding the judgement $\vdash_\Sigma [P/X](\Gamma)$ as a new premiss.

Another result we need for the admissibility proofs is how substitution effects objects, types, kinds and goals: This lemma is necessary for the weakening proof: The lemma is used only in the case case: We have to make sure, that weakening still works, even after applying the strict substitution Θ to the context of some of the premisses. The lemma would be easier to prove, if we work with rule 3.5, but since the proof is not much more difficult with rule 3.4, we simply proof the more complex lemma. The proof of the lemma is a mutual induction over ten judgments simultaneously. Later on we will have to prove to more lemmas, which are of a similar form.

Lemma 4.15 (Substitutions effects) *Let D be a data formula, P a program, K a kind, M an object and A_P an atomic type, A_G a goal type and A_D a data type. Let σ be a strict substitution, $Free(\sigma) \cap sup(\Gamma) = \emptyset$ and Λ be a context with $\vdash \Lambda$ ctx which introduces the new variables used in σ. Let $\Gamma' = \Lambda, [\sigma](\Gamma)$ and $\Delta' = [\sigma](\Delta)$. Then for all Γ meta context and Δ object context:*

$$\begin{array}{lcl}
\Gamma; \Delta \vdash_\Sigma K \; kind & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](K) \; kind \\
\Gamma; \Delta \vdash_\Sigma A_P : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) \; [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma A_G : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_G) \; [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma A_D : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_D) \; [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma M : A_P & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) \; [\sigma](A_P)
\end{array}$$

$$\begin{array}{rcl}
\Gamma; \Delta \vdash_\Sigma M : A_G & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M)\ [\sigma](A_G) \\
\Gamma; \Delta \vdash_\Sigma M : A_D & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M)\ [\sigma](A_D) \\
\Gamma \vdash_\Sigma D\ data & \Rightarrow & \Gamma' \vdash_\Sigma [\sigma](D)\ data \\
\Gamma \vdash_\Sigma P \in G & \Rightarrow & \Gamma' \vdash_\Sigma [\sigma](P) \in [\sigma](G) \\
\vdash_\Sigma \Gamma\ ctx & \Rightarrow & \vdash_\Sigma \Gamma'\ ctx
\end{array}$$

Proof: The proof goes by simultaneous induction over the structure of the assumptions. For the base cases you will need lemma 3.25, especially in the case 3.5. The strictness condition is needed for the cases LΠ, LΠV and L$\Pi\Sigma$, L$\Pi\Sigma$V. In the case case, we have to choose a new $\eta' = \Theta \circ \eta$. □

This lemma is needed when we want to prove the admissibility of weakening.

Lemma 4.16 (Weakening) *Let $\mathcal{D} :: \Gamma_1, \Gamma_2 \vdash_\Sigma P \in G$, $\mathcal{E} :: \Gamma_1 \vdash_\Sigma D'$ data and $X' \notin dom(\Gamma_1, \Gamma_2)$, then $\mathcal{D}[\Gamma_1 \bigvee X' \in D'] :: \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma P \in G$ where X' is new meta variable and D' is a data formula, depending only on variables in Γ_1.*

Proof: The proof is done by induction over the derivation $\mathcal{D}$. We describe two cases here, the other cases are described in appendix C.

Case: R $\rightarrow$: By assumption we have a sequent of the form $\Gamma_1, \Gamma_2, X \in D \vdash_\Sigma P \in G$, it is clear, that the induction hypothesis is applicable. Its application yields: $\Gamma_1, X' \in D', \Gamma_2, X \in D \vdash_\Sigma P \in G$. but now the premiss for the rule R $\rightarrow$ is still fulfilled, and we can apply it to obtain: $\Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma (\texttt{fun}\ X.P) \in D \rightarrow G$.

Case: case: The case rule is a little bit more complicated. By assumptions we have a derivation for $\Gamma_1, \Gamma_2 \vdash P \in \overline{A_G}$. By application of the induction hypothesis, we obtain $\Gamma_1, X' \in D', \Gamma_2 \vdash P \in \overline{A_G}$. Note now, that we still can use the same η for the rule, since A_G didn't change. Take the i-th premiss of the rule: By lemma 3.32, we know that $\vdash_\Sigma \Lambda^{(i)}$ ctx and $\Theta^{(i)}$ strict. Therefore $\Lambda^{(i)}, [\Theta^{(i)}](\Gamma_1) \vdash [\Theta^{(i)}](D')$ data by lemma 4.15. By applying the induction hypothesis, we obtain

$$\Lambda^{(i)}, [\Theta^{(i)}](\Gamma_1), X' \in [\Theta^{(i)}](D'), [\Theta^{(i)}](\Gamma_2) \vdash_\Sigma P^{(i)} \in [\Theta^{(i)}](G)$$

This holds for all i, therefore we can apply the rule case to obtain

$$\Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma \left(\begin{array}{ll} \texttt{case}\ P\ \texttt{of} & \\ & \overline{c_1\ \underline{Y_1^{(1)}}..\underline{Y_{m_1}^{(1)}} \Rightarrow [\eta](P^{(1)})} \\ & \ldots \\ \mid & \overline{c_n\ \underline{Y_1^{(n)}}..\underline{Y_{m_n}^{(n)}} \Rightarrow [\eta](P^{(n)})} \end{array} \right) \in G$$

□

We now address the problem of the admissibility of the contraction rule. The property of contraction becomes necessary in the proof of the local reduction theorem, the goal of this section. Contraction means, that if there are two meta variables declared in the context of the same meta formula, all occurrences of the latter may be replaced by the former one. This rule is essential for classical and intuitionistic proof systems.

For contraction we want to prove, that whenever $\mathcal{D}$ is a derivation of $\Gamma_1, U \in H, \Gamma_2, V \in H, \Gamma_3 \vdash_\Sigma P \in G$ we can find a derivation for $\Gamma_1, U \in H, \Gamma_2, [U/V](\Gamma_3) \vdash_\Sigma [U/V](P) \in [U/V](G)$. When looking at the rule set, we see that the typing rules for programs refer to the judgment for well-formed contexts. Furthermore, the typing rules for well-formed contexts refer to the judgment $\Gamma \vdash_\Sigma D$ data. The contraction lemma can only be proven by a complicated mutual inductive argument.

If we would work in a system with rule (3.5), the proof of the next theorem would be less complex, but even with the rule (3.4), the result is fairly easy to show. We prove the contraction theorem by mutual induction very similar to the proof of lemma 4.15.

Lemma 4.17 (Contraction:) *Let D, D' be data formulae, K a kind, A_P an atomic type, A_G a goal type, A_D a data type, M an object and P a program. Then the following holds: For all meta contexts $\Gamma_1, \Gamma_2, \Gamma_3$, and for all object context Δ: Let $\Gamma = \Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3$, and $\Gamma' = \Gamma_1, U \in D', \Gamma_2, [U/V](\Gamma_3)$ and let $\Delta' = [U/V](\Delta)$ and $\sigma = [U/V]$. Then we have:*

$$\begin{aligned}
\Gamma; \Delta \vdash_\Sigma K \textit{ kind} &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](K) \textit{ kind} && (4.1)\\
\Gamma; \Delta \vdash_\Sigma A_P : K &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) : [\sigma](K) && (4.2)\\
\Gamma; \Delta \vdash_\Sigma A_G : K &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_D) : [\sigma](K) && (4.3)\\
\Gamma; \Delta \vdash_\Sigma A_D : K &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_G) : [\sigma](K) && (4.4)\\
\Gamma; \Delta \vdash_\Sigma M : A_P &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_P) && (4.5)\\
\Gamma; \Delta \vdash_\Sigma M : A_D &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_D) && (4.6)\\
\Gamma; \Delta \vdash_\Sigma M : A_G &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_G) && (4.7)\\
\Gamma \vdash_\Sigma D \textit{ data} &\Rightarrow \Gamma' \vdash_\Sigma [\sigma](D) \textit{ data} && (4.8)\\
\Gamma \vdash_\Sigma P \in G &\Rightarrow \Gamma' \vdash_\Sigma [\sigma](P) \in [\sigma](G) && (4.9)\\
\vdash_\Sigma \Gamma \textit{ ctx} &\Rightarrow \vdash_\Sigma \Gamma' \textit{ ctx} && (4.10)
\end{aligned}$$

Proof: by mutual induction. The detailed proof can be found in appendix C. □

This lemma brings us directly the desired result:

Theorem 4.18 (Admissibility of Contraction) *: If*

$$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3 \vdash_\Sigma P \in G$$

then

$$\Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](P) \in [\sigma](G)$$

with $\sigma = [U/V]$. If Γ_3 is pure, P is pure and G is pure, then $[\sigma](\Gamma_3), [\sigma](P), [\sigma](G)$ are pure.

Proof: The first result is a restatement of (4.9), the second follows directly, because σ is strictly pure. □

4.2.2 Substitution lemma

The next lemma we introduce is the substitution lemma. It is not a logical rule like weakening and contraction, but it represents in a way the connection between LF and the meta logic.

Assume we have a derivation of $\Gamma \vdash_\Sigma P \in G$. This can be interpreted from a programming point of view, as: When all variables in Γ are instantiated with some programs, P will compute a proof term of G. It could be that we have a declaration of the form $X \in \overline{A}$ in the context Γ. Under the interpretation this reads, for a given LF constant $c : A$, we can plug $\overline{c}$ into the variable positions in P. The evaluation of P will provide a proof object of G.

In the next section we are concerned with local reductions. One local reduction which might occur is exactly of this form: Assume we are given an object $c : A$, and a derivation of $\Gamma_1, X \in \overline{A}, \Gamma_2 \vdash_\Sigma P \in G$, then we would expect the existence of a derivation $\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](P) \in [\sigma](G)$ where $\sigma = [\overline{c}/X]$.

The substitution lemma shows, that this holds. We can assume, that A are closed with respect to object and meta variables.

Lemma 4.19 (Substitution property:) *Let D be data formulae, K a kind, A_P an atomic type, A_G goal type, A_D, A data types, M object and P a program. For $c : A \in \Sigma$ the following holds: For all meta contexts Γ_1, Γ_2, and for all object context Δ: Let $\Gamma = \Gamma_1, Z \in \overline{A}, \Gamma_2$, and $\Gamma' = \Gamma_1, [\overline{c}/Z](\Gamma_2)$ and let $\Delta' = [\overline{c}/Z](\Delta)$ and $\sigma = [\overline{c}/Z]$. Then we have:*

$$\begin{aligned}
\Gamma; \Delta \vdash_\Sigma K\ kind &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](K)\ kind && (4.11)\\
\Gamma; \Delta \vdash_\Sigma A_P : K &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) : [\sigma](K) && (4.12)\\
\Gamma; \Delta \vdash_\Sigma A_G : K &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_G) : [\sigma](K) && (4.13)\\
\Gamma; \Delta \vdash_\Sigma A_D : K &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_D) : [\sigma](K) && (4.14)\\
\Gamma; \Delta \vdash_\Sigma M : A_P &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_P) && (4.15)\\
\Gamma; \Delta \vdash_\Sigma M : A_G &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_G) && (4.16)\\
\Gamma; \Delta \vdash_\Sigma M : A_D &\Rightarrow \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_D) && (4.17)\\
\Gamma \vdash_\Sigma D\ data &\Rightarrow \Gamma' \vdash_\Sigma [\sigma](D)\ data && (4.18)\\
\Gamma \vdash_\Sigma P \in G &\Rightarrow \Gamma' \vdash_\Sigma [\sigma](P) \in [\sigma](G) && (4.19)\\
\vdash_\Sigma \Gamma\ ctx &\Rightarrow \vdash_\Sigma \Gamma'\ ctx && (4.20)
\end{aligned}$$

Proof: by mutual induction. The detailed proof can be found in appendix C. □

A reformulation of this lemma gives us the desired substitution lemma.

Theorem 4.20 (Admissibility of Substitution) *: If $\Gamma_1, Z \in \overline{A}, \Gamma_2 \vdash_\Sigma P \in G$, and $c : A \in \Sigma$, then*

$$\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](P) \in [\sigma](G)$$

with $\sigma = [\overline{c}/Z]$. If Γ_2 is pure, P is pure and G is pure, then $[\sigma](\Gamma_2), [\sigma](P), [\sigma](G)$ are pure.

Proof: The first result is a restatement of (4.19), the second follows directly, because σ is strictly pure. □

4.2.3 Local reductions of MLF

In this section we show local reductions for MLF. The problem of local reduction can be circumscribed as follows: Assume there there are two derivations:

$$\begin{array}{c}\mathcal{D}_1\\ \Gamma_1 \vdash_\Sigma P_1 \in C\end{array} \quad \text{and} \quad \begin{array}{c}\mathcal{D}_2\\ \Gamma_2 \vdash_\Sigma P_2 \in G\end{array}$$

The derivation $\mathcal{D}_1$ constructs a proof term P_1, and the derivation $\mathcal{D}_2$ "wants" to consume P_1. There are different ways, of how this consumption may take place: One possibility is to cut $\mathcal{D}_1$ with $\mathcal{D}_2$. To do so, Γ_2 must be equal to $\Gamma_1, X \in C, \Gamma_2'$. We write

$$\begin{array}{c}\mathcal{D}_1\\ \Gamma_1 \vdash_\Sigma P_1 \in C\end{array} \bigotimes \begin{array}{c}\mathcal{D}_2\\ \Gamma_1, X \in C, \Gamma_2' \vdash_\Sigma P_2 \in G\end{array}$$

to describe this cut operation.

But the cut operation is not the only rule, which has this consumption property: Another rule is the **case** rule. The rule is restricted to C being an embedded type A_G. On the other side, there is not only one derivation $\mathcal{D}_2$ but many — one derivation for every form of the outermost constructor of P.

The aim of a local reduction theorem is to show that derivations can be locally reduced. The **cut** rule and the **case** rule must be displacable from their positions, but a derivation with the same conclusion can still be found.

In this work we present a local reduction theorem only for the **cut** case. It seems quite possible, that the theorem can be extended to the **case** case — but such an examination would be beyond the scope of this thesis.

We can put the proof term P_1 into the center of our consideration. In our proof of local reduction, we examine the different forms a program P_1 can take. Since we decided to omit **case**, we have to exclude embedded LF objects as possible proof terms P_1. The argument for all other forms is very close to the essential cases in the cut theorem in [Pfe94c, Pfe94b]. Here is the theorem:

Theorem 4.21 (Local reductions in MLF:) *If $\mathcal{D} :: \Gamma_1 \vdash P' \in C$, $P' \neq \overline{M}$ and $\mathcal{E} :: \Gamma_1, Z \in C, \Gamma_2 \vdash P \in G$ then there is a derivation $\mathcal{F}$, s.t. $\mathcal{F} :: \Gamma_1, \sigma(\Gamma_2) \vdash \sigma(P) \in \sigma(G)$ with $\sigma = [P'/Z]$.*

Proof: The detailed proof can be found in appendix C. □

Chapter 5

The example revisited

In section 2.1 we introduced a toy programming language $\mathcal{T}$ and its natural and operational semantics. We described its representation in Elf and Coq. We showed the equivalence of both semantical notions in the equivalence theorem 2.2. When using Coq as a representation mechanism we took advantage of its inductive reasoning component. We proved the append lemma 2.7, the subcomputation lemma 2.1 and the equivalence theorem 2.2 using the Coq proof engine. The representation of $\mathcal{T}$ and both semantical notions in LF and Elf was straightforward. In this chapter we discuss how MLF can be used to prove the append lemma, the subcomputation lemma and the equivalence theorem.

5.1 Append Lemma

We have seen in the definition of a multi step relation the the concatenation of a single step transition and a multi step transition results in a multi step transition. We have also seen in the proof of the subcomputation lemma 2.1, that a more sophisticated concept of concatenation is required: It is not enough to extend a trace by one single step transition up front. It has to be shown, that two traces can be concatenated to a new longer trace as long as the final state of the first coincides with the start state of the second. This property is represented by the lemma append:

Lemma 2.7 *(append) For every two traces $T : S \overset{*}{\Rightarrow} S'$ and $T' : S' \overset{*}{\Rightarrow} S''$ there exists a trace $R : S \overset{*}{\Rightarrow} S''$.*

The objective of this section is to present a derivation of the append lemma in MLF. To do so, we transform the lemma into a sequent in MLF: We start with an empty context. The formula can be represented as a MLF goal formula:

$$\forall S : state.\ \forall S' : state.\ \forall S'' : state.\ \forall T : \underline{S} \overset{*}{\Rightarrow} \underline{S'}.\ \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}}$$

We obtain the following sequent:

$$\cdot$$

$$\vdash_\Sigma \quad \mathcal{P}_1 \in \forall S : state.\ \forall S' : state.\ \forall S'' : state.\ \forall T : \underline{S} \overset{*}{\Rightarrow} \underline{S'}.\ \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}}$$

We will now give a derivation for this sequent in MLF: We apply the MLF typing rules in a bottom to top fashion. We will not construct the proof terms explicitly. The proof terms are getting rather big, because we will very often apply the L$\forall$ rule. Therefore instead of writing the proof terms, we use $\mathcal{P}_0, \mathcal{P}_1 \ldots$. as a variable notation for omitted proof terms.

Proof: We will now begin with the presentation of the proof in MLF: The first thing to do, is to provide the induction hypothesis. This is done by using the **rec** rule. We will omit the proof, that $F \downarrow \mathcal{P}_0$. It will become clear, that there is a suitable termination ordering. The proof is done by structural induction.

$$\begin{array}{ll} & F \in \forall S : \mathit{state}.\ \forall S' : \mathit{state}.\ \forall S'' : \mathit{state}.\ \forall T : \underline{S} \overset{*}{\Rightarrow} \underline{S'}.\ \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}} \\ \vdash_\Sigma & \mathcal{P}_0 \in \forall S : \mathit{state}.\ \forall S' : \mathit{state}.\ \forall S'' : \mathit{state}.\ \forall T : \underline{S} \overset{*}{\Rightarrow} \underline{S'}.\ \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

The contexts will contain many variable declarations. Instead of listing them all, we abbreviate the contexts, by omitting the corresponding data formulae for all those variable which do not participated in a rule application. Because of the form of the goal formula, we can can now apply the rule R$\forall$ four times and obtain:

$$\begin{array}{ll} & F, S \in \overline{\mathrm{state}}, S' \in \overline{\mathrm{state}}, S'' \in \overline{\mathrm{state}}, T \in \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S'}} \\ \vdash_\Sigma & \mathcal{P}_1 \in \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

In the informal proof we showed the append lemma by induction over the first derivation. Consequently, the next rule to apply is the **case** rule and not the R$\forall$ rule. The application of the **case** rule is triggered by the axiom derivation

$$\begin{array}{ll} & F, S, S', S'', T \in \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S'}} \\ \vdash_\Sigma & T \in \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S'}} \end{array}$$

We now construct the set $\mathrm{Ind}_{\Sigma,\Gamma}(T, (\underline{S} \overset{*}{\Rightarrow} \underline{S'}))$. It has two elements:

$$\begin{array}{l} \langle (D \in \mathrm{state}), \\ \ (D/S, D/S', \overline{\mathrm{id}}/T) \rangle \\ \langle (D \in \overline{\mathrm{state}}, D' \in \overline{\mathrm{state}}, D'' \in \overline{\mathrm{state}}, E \in \overline{\underline{D} \Rightarrow \underline{D'}}, E' \in \overline{\underline{D'} \overset{*}{\Rightarrow} \underline{D''}}), \\ \ (D''/S, D/S', \overline{\tilde{}\ \ \underline{D}\ \underline{D'}\ \underline{D'''}\ \underline{E}\ \underline{E'}}/T) \rangle \end{array}$$

We abbreviate both entries of $\mathrm{Ind}_{\Sigma,\Gamma}(T, (\underline{S} \overset{*}{\Rightarrow} \underline{S'}))$ for now by $\langle \Lambda^{(1)}, \Theta^{(1)} \rangle$ and $\langle \Lambda^{(2)}, \Theta^{(2)} \rangle$. For a successful application of the **case** statement we have to prove two judgments. The first results from extending the current context by $\Lambda^{(1)}$, and applying the substitution $\Theta^{(1)}$ to the current context and the current goal formula. Similarly, the second results from using $\langle \Lambda^{(2)}, \Theta^{(2)} \rangle$ instead of $\langle \Lambda^{(1)}, \Theta^{(1)} \rangle$.

Case: $\langle \Lambda^{(1)}, \Theta^{(1)} \rangle$: The current sequent has the following form

$$\begin{array}{ll} & D \in \mathrm{state}, F, S'' \\ \vdash_\Sigma & \mathcal{P}_2 \in \forall T' : \underline{D} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

One application of R∀ yields:

$$\begin{array}{ll} & D, F, S'', T' \in \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \\ \vdash_\Sigma & T' : \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

The first proof branch is closed using id again. it corresponds to the base case of the induction.

Case: $\langle \Lambda^{(2)}, \Theta^{(2)} \rangle$: The current sequent has the following form:

$$\begin{array}{ll} & D \in \overline{\text{state}}, D' \in \overline{\text{state}}, D'' \in \overline{\text{state}}, E \in \overline{\underline{D} \Rightarrow \underline{D'}}, E' \in \overline{\underline{D'} \overset{*}{\Rightarrow} \underline{D''}}, \\ & F, S'' \\ \vdash_\Sigma & \mathcal{P}_3 \in \forall T' : \underline{D''} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

First we apply R∀: On the right side of the sequent symbol remains an embedded LF type. The objective is it now, to use the context and the signature to provide a proof object of this type.

$$\begin{array}{ll} & D, D', D'', E, E', F, S'', T' \in \overline{\underline{D''} \overset{*}{\Rightarrow} \underline{S''}} \\ \vdash_\Sigma & \mathcal{P}_4 \in \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

The only way to do so, is to apply the induction hypothesis: We expect a proof term for a trace, which ends in the state S''. The induction hypothesis is applied to five parameters: D' D'' S'' E' T', i.e. five applications of L∀ yield:

$$\begin{array}{ll} & D, D', D'', E \in \overline{\underline{D} \Rightarrow \underline{D'}}, E', F, S'', T', R_5 \in \overline{\underline{D'} \overset{*}{\Rightarrow} \underline{S''}} \\ \vdash_\Sigma & \mathcal{P}_5 \in \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

From looking at this rule, it is obvious that E and R_5 have to be concatenated. Since E is a single step transition, it is enough to apply rule LΠΣ with constant ˜ and parameter D. We obtain the an assumption S_1 of the embedded partially instantiated type of ˜ :

$$\begin{array}{ll} & D, D', D'', E \in \overline{\underline{D} \Rightarrow \underline{D'}}, E', F, S'', T', R_5 \in \overline{\underline{D'} \overset{*}{\Rightarrow} \underline{S''}} \\ & S_1 \in \overline{\Pi St' : \text{state}.\ \Pi St'' : \text{state}.\ \underline{D} \Rightarrow \underline{St'} \to \underline{St'} \overset{*}{\Rightarrow} \underline{St''} \to \underline{D} \overset{*}{\Rightarrow} \underline{St''}} \\ \vdash_\Sigma & \mathcal{P}_6 \in \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

S_1 represents an embedded function type, which must be applied to four more parameters to represent an embedded objects. To do so, LΠ has to be applied four times to D' S'' E R_5. We neither give the intermediate steps nor the intermediate additional assumptions: The current sequent in the proof is

$$\begin{array}{ll} & D, D', D'', E, E', F, S'', T', R_5, S_1, S_5 \in \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \\ \vdash_\Sigma & S_5 \in \overline{\underline{D} \overset{*}{\Rightarrow} \underline{S''}} \end{array}$$

This sequent can be closed using the axiom rule id.

□

5.2 Subcomputation Lemma

We address now the proof of the subcomputation lemma from section 2.1. The subcomputation lemma is a generalization of one direction of the equivalence theorem. Here is the formulation of the lemma — this time directly formulated in terms of LF type theory:

Lemma 2.1 *Let K be an environment, E be an expression and V be a value. If D is an object in* (feval K E W) *then for all H environment stack, P program and S value stack we can find a proof term E in*

$$\text{st } (H;;K)\ (\text{ev } E\&P)\ S \overset{*}{\Rightarrow} \text{st } H\ P\ (S;W)$$

We know, that we will apply the append lemma in the proof — this is exactly why we proved the append lemma in MLF. Lemma application is done by applying a cut. For the proof, we have to assume that we have the lemma handy. We will therefore prove the subcomputation lemma under the assumption, that lemma append is available. This is done by declaring the variable *Append* of type

$$\forall S : \textit{state}.\ \forall S' : \textit{state}.\ \forall S'' : \textit{state}.\ \forall T : \underline{S} \overset{*}{\Rightarrow} \underline{S'}.\ \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}}$$

and putting this new declaration into the context. The representation of the the subcomputation lemma as a sequent in MLF is as follows:

$$\begin{array}{ll} & \textit{Append} \in \forall S : \textit{state}.\ \forall S' : \textit{state}.\ \forall S'' : \textit{state}.\ \forall T : \underline{S} \overset{*}{\Rightarrow} \underline{S'}.\ \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}} \\ \vdash_\Sigma & \mathcal{P}_0 \in \forall K : \text{env}.\ \forall E : \text{exp}.\ \forall W : \text{val}.\ \forall D : \text{feval } \underline{K}\ \underline{E}\ \underline{W}. \\ & \forall H : \text{envstack}.\ \forall P : \text{program}.\ \forall S : \text{env}.\ \overline{\text{st } (\underline{H};;\underline{K})\ (\text{ev } \underline{E}\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W})} \end{array}$$

Proof: As in the proof of the append lemma we provide the induction hypothesis as a first step in the proof.

$$\begin{array}{ll} & \textit{Append}, \\ & F \in \forall K : \text{env}.\ \forall E : \text{exp}.\ \forall W : \text{val}.\ \forall D : \text{feval } \underline{K}\ \underline{E}\ \underline{W}. \\ & \forall H : \text{envstack}.\ \forall P : \text{program}.\ \forall S : \text{env}.\ \overline{\text{st } (\underline{H};;\underline{K})\ (\text{ev } \underline{E}\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W})} \\ \vdash_\Sigma & \mathcal{P}_1 \in \forall K : \text{env}.\ \forall E : \text{exp}.\ \forall W : \text{val}.\ \forall D : \text{feval } \underline{K}\ \underline{E}\ \underline{W}. \\ & \forall H : \text{envstack}.\ \forall P : \text{program}.\ \forall S : \text{env}.\ \overline{\text{st } (\underline{H};;\underline{K})\ (\text{ev } \underline{E}\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W})} \end{array}$$

The objective is now to decompose the goal formula. We propose to apply R∀ four times. The variable D is the variable we want to perform induction over.

$$\begin{array}{ll} & Append, F, K \in \overline{\text{env}}, E \in \overline{\text{exp}}, W \in \overline{\text{val}}, D \in \overline{\text{feval}\ \underline{K}\ \underline{E}\ \underline{W}} \\ \vdash_\Sigma & \mathcal{P}_2 \in \forall H : \text{envstack}. \forall P : \text{program}. \forall S : \text{env}. \overline{\text{st}\ (\underline{H};;\underline{K})\ (\text{ev}\ \underline{E}\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W})} \end{array}$$

Induction is performed by case distinction over $\overline{\text{feval}\ \underline{K}\ \underline{E}\ \underline{W}}$. First we construct the set $\text{Ind}_{\Sigma,\Gamma}(D, \text{feval}\ \underline{K}\ \underline{E}\ \underline{W})$: it contains four elements:

$$\begin{array}{l} \{\langle(K_0 \in \overline{\text{env}}, W_0 \in \overline{\text{val}}), \\ \quad (K_0/K, \overline{1}/E, W_0/W, \overline{\text{ev1}\ \underline{K_0}\ \underline{W_0}}/D)\rangle, \\ \langle(K_0 \in \overline{\text{env}}, E_0 \in \overline{\text{exp}}, W_0 \in \overline{\text{val}}, W_0' \in \overline{\text{val}}, D_0 \in \overline{\text{feval}\ \underline{K_0}\ \underline{E_0}\ \underline{W_0}}), \\ \quad (\overline{(\underline{K_0};\underline{W_0'})}/K, \overline{\underline{E_0}\uparrow}/E, W_0/W, \overline{(\text{ev}\uparrow\ \underline{K_0}\ \underline{E_0}\ \underline{W_0}\ \underline{W_0'}\ \underline{D_0})}/D)\rangle, \\ \langle(K_0 \in \overline{\text{env}}, E_0 \in \overline{\text{exp}}), \\ \quad (K_0/K, \overline{\text{lam}\ \underline{E_0}}/E, \overline{\text{clo}\ \underline{K_0}\ (\text{lam}\ \underline{E_0})}/W, \overline{\text{evlam}\ \underline{K_0}\ \underline{E_0}}/D)\rangle, \\ \langle(K_1 \in \overline{\text{env}} E_2 \in \overline{\text{exp}}, K_1' \in \overline{\text{env}}, E_2' \in \overline{\text{exp}}, E_3 \in \overline{\text{exp}}, W_3 \in \overline{\text{val}}, W_1 \in \overline{\text{val}}, \\ \quad D_1 \in \overline{\text{feval}\ \underline{K_1}\ \underline{E_2}\ (\text{clo}\ \underline{K_1'}\ (\text{lam}\ \underline{E_2'}))}, D_2 \in \overline{\text{feval}\ \underline{K_1}\ \underline{E_3}\ \underline{W_3}}, \\ \quad D_3 \in \overline{\text{feval}\ (\underline{K_1'};\underline{W_3})\ \underline{E_2'}\ \underline{W_1}}), \\ \quad (K_0/K, \overline{\text{app}\ \underline{E_2}\ \underline{E_3}}/E, W_0/W, \overline{\text{evapp}\ \underline{K_1}\ \underline{E_2}\ \underline{K_1'}\ \underline{E_2'}\ \underline{E_3}\ \underline{W_3}\ \underline{W_1}\ \underline{D_1}\ \underline{D_2}\ \underline{D_3}}/D)\rangle\} \end{array}$$

The application of **case** leads now to four new judgements defined by current sequent and each entry $\langle\Lambda^{(i)}, \Theta^{(i)}\rangle \in \text{Ind}_{\Sigma,\Gamma}(D, \text{feval}\ \underline{K}\ \underline{E}\ \underline{W})$. We address the derivation of each of those four judgments:

Case: ev1: The first case corresponds to the base case. The current sequent is extended by $\Lambda^{(1)}$ and the substitution $\Theta^{(1)}$ is applied to the current context and the current goal formula:

$$\begin{array}{ll} & K_0 \in \overline{\text{env}}, W_0 \in \overline{\text{val}}, Append, F \\ \vdash_\Sigma & \mathcal{P}_3 \in \forall H : \text{envstack}. \forall P : \text{program}. \forall S : \text{env}. \\ & \overline{\text{st}\ (\underline{H};;\underline{K})\ (\text{ev}\ 1\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \end{array}$$

The goal formula can be further decomposed by applying R$\forall$ three times:

$$\begin{array}{ll} & K_0, W_0, Append, F, H \in \overline{\text{envstack}}, P \in \overline{\text{program}}, S \in \overline{\text{env}} \\ \vdash_\Sigma & \mathcal{P}_4 \in \overline{\text{st}\ (\underline{H};;(\underline{K_0};\underline{W_0}))\ (\text{ev}\ 1\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \end{array}$$

For this goal we can find a constant in the signature, which yields an instance of the desired type: c_1 H K_0 W_0 P S. To obtain the proof object, we have to apply L$\Pi\Sigma$ once — the parameter is H, and four times LΠ.

$$\begin{array}{ll} & K_0, W_0, Append, F, H \in \overline{\text{envstack}}, P \in \overline{\text{program}}, S \in \overline{\text{env}}, \\ & R_5 \in \overline{\text{st}\ (\underline{H};;(\underline{K_0};\underline{W_0}))\ (\text{ev}\ 1\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \\ \vdash_\Sigma & R_5 \in \overline{\text{st}\ (\underline{H};;(\underline{K_0};\underline{W_0}))\ (\text{ev}\ 1\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \end{array}$$

Finally we can close this branch with **id**: The first base case is proven.

Case: ev$\uparrow$: The next judgment is the result of applying the second element of $\mathrm{Ind}_{\Sigma,\Gamma}(D, \mathrm{feval}\ \underline{K}\ \underline{E}\ \underline{W})$ to the current sequent. We obtain the sequent:

$$\begin{array}{ll} & K_0 \in \overline{\mathrm{env}}, E_0 \in \overline{\mathrm{exp}}, W_0 \in \overline{\mathrm{val}}, W_0' \in \overline{\mathrm{val}}, D_0 \in \overline{\mathrm{feval}\ \underline{K_0}\ \underline{E_0}\ \underline{W_0}}, \mathit{Append}, F \\ \vdash_\Sigma & \mathcal{P}_5 \in \forall H : \mathrm{envstack}.\ \forall P : \mathrm{program}. \\ & \overline{\forall S : \mathrm{env}.\ \mathrm{st}\ (\underline{H};;(\underline{K_0};\underline{W_0'}))\ (\mathrm{ev}\ (\underline{E_0}\uparrow)\&\underline{P})\ S \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \end{array}$$

As in the last case, the goal formula is decomposed by three times applying R$\forall$, we obtain

$$\begin{array}{ll} & K_0, E_0, W_0, W_0', D_0, \mathit{Append}, F, \\ & H \in \overline{\mathrm{envstack}}, P \in \overline{\mathrm{program}}, S \in \overline{\mathrm{env}} \\ \vdash_\Sigma & \mathcal{P}_6 \in \overline{\mathrm{st}\ (\underline{H};;(\underline{K_0};\underline{W_0'}))\ (\mathrm{ev}\ (\underline{E_0}\uparrow)\&\underline{P})\ S \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \end{array}$$

Examining this sequent we see, that the induction hypothesis F can be applied to D_0 to obtain a trace which ends in the desired state. To apply the induction hypothesis, we have to instantiate all universal quantifiers of F with a list of parameters: $K_0\ E_0\ W_0\ D_0\ H\ P\ S$: After seven applications of L$\forall$ we obtain

$$\begin{array}{ll} & K_0, E_0, W_0, W_0', D_0, \mathit{Append}, F, H, P, S \\ & R_7 \in \overline{\mathrm{st}\ (\underline{H};;\underline{K_0})\ (\mathrm{ev}\ \underline{E_0}\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \\ \vdash_\Sigma & \mathcal{P}_7 \in \overline{\mathrm{st}\ (\underline{H};;(\underline{K_0};\underline{W_0'}))\ (\mathrm{ev}\ (\underline{E_0}\uparrow)\&\underline{P})\ S \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \end{array}$$

By using L$\Pi\Sigma$ to access c_ $\uparrow$ with the argument H and by applying LΠ with the newly generated function five times to the parameters $K_0\ W_0'\ E_0\ P\ S$ we obtain a proof term S_6 of a trace, which starts in the same state as the embedded goal formula, and ends in the state where R_7 starts.

$$\begin{array}{ll} & K_0, E_0, W_0, W_0', D_0, \mathit{Append}, F, H, P, S \\ & R_7 \in \overline{\mathrm{st}\ (\underline{H};;\underline{K_0})\ (\mathrm{ev}\ \underline{E_0}\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \\ & S_6 \in \mathrm{st}\ (\underline{H};;(\underline{K_0};\underline{W_0'}))\ (\mathrm{ev}\ (\underline{E_0}\uparrow)\&\underline{P})\ \underline{S} \Rightarrow \mathrm{st}\ (\underline{H};;\underline{K_0})\ (\mathrm{ev}\ \underline{E_0}\&\underline{P})\ \underline{S} \\ \vdash_\Sigma & \mathcal{P}_8 \in \overline{\mathrm{st}\ (\underline{H};;(\underline{K_0};\underline{W_0'}))\ (\mathrm{ev}\ (\underline{E_0}\uparrow)\&\underline{P})\ S \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})} \end{array}$$

All what remains to do is to concatenate both traces, witnessed by S_6 and R_7 to obtain the desired trace. We do this, by constructing a new trace using the trace constructor $\tilde{}$. This is done by using L$\Pi\Sigma$ once and LΠ four times with a list of parameters: $\overline{\mathrm{st}\ (\underline{H};;(\underline{K_0};\underline{W_0'}))\ (\mathrm{ev}\ (\underline{E_0}\uparrow)\&\underline{P})\ \underline{S}}, \overline{\mathrm{st}\ (\underline{H};;\underline{K_0})\ (\mathrm{ev}\ \underline{E_0}\&\underline{P})\ \underline{S}}, \overline{\mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})}, S_6, R_7$: We do not show how the composite objects are proven type correct. Here is the final sequent:

$$\begin{array}{ll}
 & K_0, E_0, W_0, W_0', D_0, \mathit{Append}, F, H, P, S, \\
 & R_7, S_6 \\
 & T_5 \in \text{st } (\underline{H};;(\underline{K_0};\underline{W_0'}))\ (\text{ev } (\underline{E_0}\uparrow)\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0}) \\ \hline
\vdash_\Sigma & T_5 \in \text{st } (\underline{H};;(\underline{K_0};\underline{W_0'}))\ (\text{ev } (\underline{E_0}\uparrow)\&\underline{P})\ S \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_0})
\end{array}$$

which can be closed using **id**. This proof branch is closed and we can address the next case.

Case: evlam: This case addresses the treatment of λ-abstraction. The sequent is obtained, by using the original sequent and applying $\langle \Lambda^{(3)}, \Theta^{(3)} \rangle$. This proof branch is therefore initialized by the following sequent.

$$\begin{array}{ll}
 & K_0 \in \overline{\text{env}}, E_0 \in \overline{\text{exp}}, \mathit{Append}, F \\
\vdash_\Sigma & \mathcal{P}_9 \in \forall H : \text{envstack}.\ \forall P : \text{program}.\ \forall S : \text{env}. \\ \hline
 & \text{st } (\underline{H};;\underline{K_0})\ (\text{ev } (\text{lam } \underline{E_0})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};(\text{clo } \underline{K_0}\ (\text{lam } \underline{E_0})))
\end{array}$$

As in the last three cases, We apply the rule R∀ three times and obtain the sequent:

$$\begin{array}{ll}
 & K_0, E_0, \mathit{Append}, F, H \in \overline{\text{envstack}}, P \in \overline{\text{program}}, S \in \overline{\text{env}} \\ \hline
\vdash_\Sigma & \mathcal{P}_{10} \in \text{st } (\underline{H};;\underline{K_0})\ (\text{ev } (\text{lam } \underline{E_0})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};(\text{clo } \underline{K_0}\ (\text{lam } \underline{E_0})))
\end{array}$$

By providing the correct parameters to c_lam — $H\ K_0\ E_0\ P\ S$ we obtain after one LΠΣ and four LΠ rule applications the sequent:

$$\begin{array}{ll}
 & K_0, E_0, \mathit{Append}, F, H, P, S, \\
 & R_5 \in \text{st } (\underline{H};;\underline{K_0})\ (\text{ev}(\text{lam } \underline{E_0})\&\underline{P})\ \underline{S} \Rightarrow \text{st } \underline{H}\ \underline{P}\ (\underline{S};(\text{clo } \underline{K_0}(\text{lam } \underline{E_0}))) \\ \hline
\vdash_\Sigma & R_5 \in \text{st } (\underline{H};;\underline{K_0})\ (\text{ev } (\text{lam } \underline{E_0})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};(\text{clo } \underline{K_0}\ (\text{lam } \underline{E_0})))
\end{array}$$

which can be closed by **id**. The third case is proven.

Case: evapp: The proof for the last case is the most complicated one. Three applications of the induction hypothesis and two applications of the append lemma are necessary to prove this branch. As usual, we obtain the initial sequent from the current one after applying $\langle \Lambda^{(4)}, \Theta^{(4)} \rangle$ to the current sequent:

$$\begin{array}{ll}
 & K_1 \in \overline{\text{env}}, E_2 \in \overline{\text{exp}}, K_1' \in \overline{\text{env}}, E_2' \in \overline{\text{exp}}, E_3 \in \overline{\text{exp}}, W_3 \in \overline{\text{val}}, W_1 \in \overline{\text{val}}, \\
 & D_1 \in \overline{\text{feval } \underline{K_1}\ \underline{E_2}\ (\text{clo } \underline{K_1'}\ (\text{lam } \underline{E_2'}))}, \\
 & D_2 \in \overline{\text{feval } \underline{K_1}\ \underline{E_3}\ \underline{W_3}}, \\
 & D_3 \in \overline{\text{feval } (\underline{K_1'};\underline{W_3})\ \underline{E_2'}\ \underline{W_1}} \\
 & \mathit{Append}, F \\
\vdash_\Sigma & \mathcal{P}_{11} \in \forall H : \text{envstack}.\ \forall P : \text{program}.\ \forall S : \text{env}. \\ \hline
 & \text{st } (\underline{H};;\underline{K_1})\ (\text{ev } (\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})
\end{array}$$

To get rid of the forall quantifiers in the goal formula, we have to apply the R$\forall$ rule three times.

$$\begin{array}{ll} & K_1, E_2, K_1', E_2', E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F \\ & H \in \overline{\text{envstack}}, P \in \overline{\text{program}}, S \in \overline{\text{env}} \\ \vdash_\Sigma & \mathcal{P}_{11}' \in \overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev } (\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})} \end{array}$$

This time, we go through the proof in a very forward directed way: We first try to generate a transition from the start state of the embedded goal type to some other state. The only way to do so is to use c_app. Since this is a constant defined in the signature, L$\Pi\Sigma$ is applied with the parameter H, to move it into the context. Then LΠ is applied five times with five more parameters K_1 E_2 E_3 P S. We obtain

$$\begin{array}{ll} & K_1, E_2, K_1', E_2', E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S \\ & U_6 \in \overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev}(\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}} \\ & \qquad \overline{\Rightarrow \text{st } (\underline{H};;\underline{K_1};;\underline{K_1})\ (\text{ev } \underline{E_2}\&\text{ev } \underline{E_3}\&\text{apply}\&\underline{P})\ \underline{S}} \\ \vdash_\Sigma & \mathcal{P}_{12} \in \overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev } (\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})} \end{array}$$

Next, we try to go from the final state of U_6 to some other state. Obviously, we have to construct a trace which starts in the final state of U_6. It is easy to see, that this trace is provided by applying the induction hypothesis to D_1, since this is the only proposition in which E_2 occurs. The application of L$\forall$ seven times with the parameters: $K_1, E_2, \overline{\text{clo } \underline{K_1'}\ (\text{lam } \underline{E_2'})}, D_1, \overline{\underline{H};;\underline{K1}}, \overline{\text{ev } \underline{E_3}\&\text{apply}\&\underline{P}}, S$ yields a proof term of the desired type: R_7.

$$\begin{array}{ll} & K_1, E_2, K_1', E_2', E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S \\ & U_6 \in \overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev}(\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}} \\ & \qquad \overline{\Rightarrow \text{st } (\underline{H};;\underline{K_1};;\underline{K_1})\ (\text{ev } \underline{E_2}\&\text{ev } \underline{E_3}\&\text{apply}\&\underline{P})\ \underline{S}} \\ & R_7 \in \overline{\text{st } (\underline{H};;\underline{K_1};;\underline{K_1})\ (\text{ev } \underline{E_2}\&\text{ev } \underline{E_3}\&\text{apply}\&\underline{P})\ \underline{S}} \\ & \qquad \overline{\overset{*}{\Rightarrow} \text{st } (\underline{H}\ ;;\underline{K_1})\ (\text{ev } \underline{E_3}\&\text{apply}\&\underline{P})\ (\underline{S};(\text{clo } \underline{K_1'}\ (\text{lam } \underline{E_2'})))} \\ \vdash_\Sigma & \mathcal{P}_{13} \in \overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev } (\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})} \end{array}$$

Now we are in the situation where we have a single step transition U_6 and a multi step transition R_7 which should be concatenated. This is done by the multi step transition constructor $\tilde{}$. To apply $\tilde{} \in \Sigma$ we have to apply L$\Pi\Sigma$ once and LΠ three times with four parameters

1. $\overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev}(\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}}$
2. $\overline{\text{st } (\underline{H};;\underline{K_1};;\underline{K_1})\ (\text{ev } \underline{E_2}\&\text{ev } \underline{E_3}\&\text{apply}\&\underline{P})\ \underline{S}}$
3. $\overline{\text{st } (\underline{H}\ ;;\underline{K_1})\ (\text{ev } \underline{E_3}\&\text{apply}\&\underline{P})\ (\underline{S};(\text{clo } \underline{K_1'}\ (\text{lam } \underline{E_2'})))}$
4. D_1

The result of this application is the following sequent. Note that S_4 stands for the newly constructed trace.

$$\begin{aligned}
&\quad K_1, E_2, K_1', E_2', E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S, U_6, R_7\\
&\quad S_4 \in \overline{\overline{\mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}(\mathrm{app}\ \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}}}\\
&\qquad\quad \overline{\overset{*}{\Rightarrow} \mathrm{st}\ (\underline{H}\ ;;\underline{K_1})\ (\mathrm{ev}\ \underline{E_3}\&\mathrm{apply}\&\underline{P})\ (\underline{S};(\mathrm{clo}\ \underline{K_1'}\ (\mathrm{lam}\ \underline{E_2'})))}\\
\vdash_\Sigma &\quad \mathcal{P}_{14} \in \overline{\mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}\ (\mathrm{app}\ \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})}
\end{aligned}$$

The next step is to construct a trace starting in the final state of S_4 and leading to some other state. This time the induction hypothesis has to be applied to D_2: This is done by applying $\mathsf{L}\forall$ rule seven times with the following parameters: $K_1, E_3, W_3, D_2, H, \overline{\mathrm{apply}\&\underline{P}}, \overline{\underline{S};(\mathrm{clo}\ \underline{K_1'}\ (\mathrm{lam}\ \underline{E_2'}))}$. The result is the sequent

$$\begin{aligned}
&\quad K_1, E_2, K_1', E_2', E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S, U_6, R_7\\
&\quad S_4 \in \overline{\overline{\mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}(\mathrm{app}\ \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}}}\\
&\qquad\quad \overline{\overset{*}{\Rightarrow} \mathrm{st}\ (\underline{H}\ ;;\underline{K_1})\ (\mathrm{ev}\ \underline{E_3}\&\mathrm{apply}\&\underline{P})\ (\underline{S};(\mathrm{clo}\ \underline{K_1'}\ (\mathrm{lam}\ \underline{E_2'})))}\\
&\quad V_7 \in \overline{\overline{\mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}\ \underline{E_3}\&\mathrm{apply}\&\underline{P})\ (\underline{S};(\mathrm{clo}\ \underline{K_1'}\ (\mathrm{lam}\ \underline{E_2'})))}}\\
&\qquad\quad \overline{\overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ (\mathrm{apply}\&\underline{P})\ (\underline{S};(\mathrm{clo}\ \underline{K_1'}\ (\mathrm{lam}\ \underline{E_2'}));\underline{W_3})}\\
\vdash_\Sigma &\quad \mathcal{P}_{15} \in \mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}\ (\mathrm{app}\ \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})
\end{aligned}$$

We observe, that there are now two multi step transitions, S_4 and V_7. We cannot used $\tilde{}$ to construct a concatenation trace: S_4 is not a single step transition. We have to use the lemma *Append*. The rule $\mathsf{L}\forall$ has to be applied five times with the following parameters:

1. $\overline{\mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}(\mathrm{app}\ \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}}$,
2. $\overline{\mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}\ \underline{E_3}\&\mathrm{apply}\&\underline{P})\ (\underline{S};(\mathrm{clo}\ \underline{K_1'}\ (\mathrm{lam}\ \underline{E_2'})))}$,
3. $\overline{\mathrm{st}\ \underline{H}\ (\mathrm{apply}\&\underline{P})\ (\underline{S};(\mathrm{clo}\ \underline{K_1'}\ (\mathrm{lam}\ \underline{E_2'}));\underline{W_3})}$,
4. S_4,
5. V_7

we obtain the new sequent with the newly concatenated trace as W_5:

$$\begin{aligned}
&\quad K_1, E_2, K_1', E_2', E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S, U_6, R_7, S_4, V_7\\
&\quad W_5 \in \overline{\overline{\mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}(\mathrm{app}\ \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}}}\\
&\qquad\quad \overline{\overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ (\mathrm{apply}\&\underline{P})\ (\underline{S};(\mathrm{clo}\ \underline{K_1'}\ (\mathrm{lam}\ \underline{E_2'}));\underline{W_3})}\\
\vdash_\Sigma &\quad \mathcal{P}_{16} \in \mathrm{st}\ (\underline{H};;\underline{K_1})\ (\mathrm{ev}\ (\mathrm{app}\ \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \mathrm{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})
\end{aligned}$$

As above, we try to construct now some trace starting in the final state of W_5 leading to some other state. We have to use the constant c_apply $\in \Sigma$. To apply the rule, we have to apply $\mathsf{L}\Pi\Sigma$ once and $\mathsf{L}\Pi$ six times with the parameters $H\ P\ S\ K_1'\ E_2'\ W_5$. The newly generated trace is called X_6

$$\begin{array}{ll}
 & K_1, E_2, K'_1, E'_2, E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S, U_6, R_7, S_4, V_7 \\
 & W_5 \in \overline{\text{st }(\underline{H};;\underline{K_1})\ (\text{ev}(\text{app }\underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}} \\
 & \qquad \overline{\overset{*}{\Rightarrow} \text{st }\underline{H}\ (\text{apply}\&\underline{P})\ (\underline{S};(\text{clo }\underline{K'_1}\ (\text{lam }\underline{E'_2}));\underline{W_3})} \\
 & X_6 \in \overline{\text{st }\underline{H}\ (\text{apply}\&\underline{P})\ (\underline{S};\text{clo }\underline{K'_1}(\text{lam }\underline{E'_2});\underline{W_3})} \\
 & \qquad \overline{\Rightarrow \text{st }(\underline{H};;(\underline{K'_1};\underline{W_3}))\ (\text{ev }\underline{E'_2}\&P)\ S} \\
\hline
\vdash_\Sigma & \mathcal{P}_{17} \in \text{st }(\underline{H};;\underline{K_1})\ (\text{ev }(\text{app }\underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st }\underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})
\end{array}$$

In this situation it is tempting to try to connect W_5 and X_6 to a trace, and then try to close the gap between the final state of X_6 and the final state of the trace described by the embedded goal formula. It turns out that this is not the best way to go. The reason is, that we can connect a one step transition with a multi step transition because of the constructor $\tilde{}$, but it is difficult to connect a multi step transition with a single step transition. Lemma *Append* guarantees that we can concatenate two multi step transitions, but at this point of our consideration, we would have to invest more reasoning to derive a similar lemma for single step and multi step traces. We will not do this. Instead we will try to construct a trace starting at the final state of X_6 and leading to the desired final state. If it turns out, that this trace is only a single step transition, we have to prove the lemma. But fortunately, it will turn out to be a multi step trace. Then, we can apply lemma *Append* again.

To bridge the gap between the final state of X_6 and the desired final state, we have to apply the induction hypothesis again. We make the observation that D_3 states something about the evaluation of E'_2. To apply the induction hypothesis F we have to use the rule $\mathsf{L}\forall$ seven times with the parameters $\overline{(\underline{K'_1};\underline{W_3})}, E'_2, W_1, D_3, H, P, S$. We obtain:

$$\begin{array}{ll}
 & K_1, E_2, K'_1, E'_2, E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S, U_6, R_7, S_4, V_7, \\
 & W_5 \in \overline{\text{st }(\underline{H};;\underline{K_1})\ (\text{ev}(\text{app }\underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}} \\
 & \qquad \overline{\overset{*}{\Rightarrow} \text{st }\underline{H}\ (\text{apply}\&\underline{P})\ (\underline{S};(\text{clo }\underline{K'_1}\ (\text{lam }\underline{E'_2}));\underline{W_3})} \\
 & X_6 \in \overline{\text{st }\underline{H}\ (\text{apply}\&\underline{P})\ (\underline{S};\text{clo }\underline{K'_1}(\text{lam }\underline{E'_2});\underline{W_3})} \\
 & \qquad \overline{\Rightarrow \text{st }(\underline{H};;(\underline{K'_1};\underline{W_3}))\ (\text{ev }\underline{E'_2}\&P)\ S} \\
 & Y_7 \in \overline{\text{st }(\underline{H};;(\underline{K'_1};\underline{W_3}))\ (\text{ev }\underline{E'_2}\&\underline{P})\ \underline{S}} \\
 & \qquad \overline{\overset{*}{\Rightarrow} \text{st }\underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})} \\
\hline
\vdash_\Sigma & \mathcal{P}_{18} \in \text{st }(\underline{H};;\underline{K_1})\ (\text{ev }(\text{app }\underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st }\underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})
\end{array}$$

All what remains to do is to concatenate X_6 and Y_7 with the constructor $\tilde{}\ \in \Sigma$. The resulting trace is a multi step transition and has to be connected to W_5. To concatenate X_6 and Y_7 we apply $\mathsf{L}\Pi\Sigma$ once and $\mathsf{L}\Pi$ four times, with the the following list of parameters:

1. $\overline{\text{st }\underline{H}\ (\text{apply}\&\underline{P})\ (\underline{S};\text{clo }\underline{K'_1}(\text{lam }\underline{E'_2});\underline{W_3})}$
2. $\overline{\text{st }(\underline{H};;(\underline{K'_1};\underline{W_3}))\ (\text{ev }\underline{E'_2}\&\underline{P})\ \underline{S}}$
3. $\overline{\text{st }\underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})}$
4. X_6
5. Y_7

The result is the following sequent: Z_5 is the new trace. This trace has to be appended to W_5:

$$\begin{array}{ll}
& K_1, E_2, K_1', E_2', E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S, U_6, R_7, S_4, V_7,\\
& W_5 \in \overline{\overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev}(\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}}}\\
& \qquad \overline{\overset{*}{\Rightarrow} \text{st } \underline{H}\ (\text{apply}\&\underline{P})\ (\underline{S};(\text{clo } \underline{K_1'}\ (\text{lam } \underline{E_2'}));\underline{W_3})}\\
& X_6, Y_7.\\
& Z_5 \in \overline{\overline{\text{st } \underline{H}\ (\text{apply}\&\underline{P})\ (\underline{S};\text{clo } \underline{K_1'}(\text{lam } \underline{E_2'});\underline{W_3})}}\\
& \qquad \overline{\overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})}\\
\vdash_\Sigma & \mathcal{P}_{19} \in \text{st } (\underline{H};;\underline{K_1})\ (\text{ev } (\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})
\end{array}$$

The final step in this proof is to concatenate W_5 and Z_5: We do this by using lemma *Append* as above, this time with the parameters:

1. $\overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev}(\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S}}$
2. $\overline{\text{st } \underline{H}\ (\text{apply}\&\underline{P})\ (\underline{S};\text{clo } \underline{K_1'}(\text{lam } \underline{E_2'});\underline{W_3})}$
3. $\overline{\text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})}$
4. W_5
5. Z_5

the final sequent has the following form:

$$\begin{array}{ll}
& K_1, E_2, K_1', E_2', E_3, W_3, W_1, D_1, D_2, D_3, \mathit{Append}, F, H, P, S, U_6,\\
& R_7, S_4, V_7, W_5, X_7, Y_7, Z_5,\\
& A_5 \in \overline{\text{st } (\underline{H};;\underline{K_1})\ (\text{ev}(\text{app } \underline{E_2}\ \underline{E_3})\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S};\underline{W_1})}\\
\vdash_\Sigma & A_5 \in \text{st } (H;;K_1)\ (\text{ev } (\text{app } E_2\ E_3)\&P)\ S \overset{*}{\Rightarrow} \text{st } H\ P\ (S;W_1)
\end{array}$$

Obviously, A_5 is of the desired embedded goal type. **id** closes this branch and completes the proof the the subcomputation lemma.

We proved the subcomputation lemma under the assumption that we have a lemma accessible, which guarantees the concatenation of to traces: the lemma append. In the proof, we do not refer explicitly to the proof of the append lemma only to the variable *Append*, which is present in the context: The proof of the subcomputation lemma was done in a non-empty context. The objective is to proof the subcomputation lemma — without any further assumptions. To do so, we have to bring the proofs of subcomputation lemma and the append lemma together. This is done by using the **cut**-rule. In section 5.1 we have seen, that it is possible to derive a proof term P of the append lemma:

$$\vdash_\Sigma \quad P \in \forall S : \mathit{state}.\ \forall S' : \mathit{state}.\ \forall S'' : \mathit{state}.\ \forall T : \underline{S} \overset{*}{\Rightarrow} \underline{S'}.\ \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}}$$

Because of the subcomputation lemma, we have a derivation of a proof term Q:

$$\begin{array}{ll} & \mathit{Append} \in \forall S : \mathit{state}.\ \forall S' : \mathit{state}.\ \forall S'' : \mathit{state}.\ \forall T : \underline{S} \overset{*}{\Rightarrow} \underline{S'}.\ \forall T' : \underline{S'} \overset{*}{\Rightarrow} \underline{S''}.\ \overline{\underline{S} \overset{*}{\Rightarrow} \underline{S''}} \\ \vdash_\Sigma & Q \in \forall K : \text{env}.\ \forall E : \text{exp}.\ \forall W : \text{val}.\ \forall D : \text{feval}\ \underline{K}\ \underline{E}\ \underline{W}. \\ & \forall H : \text{envstack}.\ \forall P : \text{program}.\ \forall S : \text{env}.\ \overline{\text{st}\ (\underline{H};;\underline{K})\ (\text{ev}\ \underline{E}\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W})} \end{array}$$

The cut rule is applicable, we obtain the following proof object for the subcomputation lemma with an empty context:

$$[P/\mathit{Append}](Q) \tag{5.1}$$

□

This concludes the presentation of the subcomputation lemma. A direct consequence from this lemma is the equivalence theorem: In the next section we give a representation of this lemma in MLF:

5.3 Equivalence Theorem

The equivalence theorem states the equivalence of the natural and the operational semantics of our language $\mathcal{T}$. The subcomputation theorem shows one direction of the theorem. In this last section we give a more elegant presentation of this one direction in MLF. We restate the theorem from section 2.1.

Theorem 2.2 (Equivalence Theorem (one direction)) *For K environment, E expression and W value: If* (feval K E W) *is inhabited, then also* $\langle(\cdot;K), E\&\mathit{done}, \cdot\rangle \overset{*}{\Longrightarrow} \langle\cdot, \mathit{done}, (\cdot;W)\rangle$

Proof: The proof of this theorem follows then quite easily: We know that the theorem is a direct consequence of the subcomputation theorem. We therefore extend the context Γ by the assumption that a subcomputation lemma is available. We prove the theorem under this assumption. Eventually, we cut this assumption out, using the original proof of the subcomputation theorem. The representation of the equivalence theorem is therefore:

$$\begin{array}{ll} & \mathit{Subcomp} \in \forall K : \text{env}.\ \forall E : \text{exp}.\ \forall W : \text{val}.\ \forall D : \text{feval}\ \underline{K}\ \underline{E}\ \underline{W}. \\ & \forall H : \text{envstack}.\ \forall P : \text{program}.\ \forall S : \text{env}.\ \overline{\text{st}\ (\underline{H};;\underline{K})\ (\text{ev}\ \underline{E}\&\underline{P})\ \underline{S} \overset{*}{\Rightarrow} \text{st}\ \underline{H}\ \underline{P}\ (\underline{S};\underline{W})} \\ \vdash_\Sigma & \mathcal{P}_0 \in \forall K : \text{env}.\ \forall E : \text{exp}.\ \forall W : \text{val}.\ \overline{\text{feval}\ K\ E\ W \to} \\ & \overline{\text{st (emptys;;}K\text{) (ev }E\text{\&done) empty} \overset{*}{\Rightarrow} \text{st emptys done (empty;}W\text{)}} \end{array}$$

As usual, we first decompose the form of the goal formula on the right. Three rule R$\forall$ is applied three times, the result is the following sequent:

$$\begin{array}{ll} & \mathit{Subcomp}, K \in \overline{\text{env}}, E \in \overline{\text{exp}}, W \in \overline{\text{val}} \\ \vdash_\Sigma & \mathcal{P}_1 \in \overline{\text{feval}\ K\ E\ W \to} \\ & \overline{\text{st (emptys;;}K\text{) (ev }E\text{\&done) empty} \overset{*}{\Rightarrow} \text{st emptys done (empty;}W\text{)}} \end{array}$$

The implication in the goal formula can be resolved using the rule R $\to$: the resulting sequent has the following form:

$$
\begin{array}{ll}
 & Subcomp, K, E, W, D \in \overline{\text{feval } K\ E\ W} \\
\vdash_\Sigma & \mathcal{P}_2 \in \overline{\text{st (emptys;;}K\text{) (ev } E\text{\&done) empty} \overset{*}{\Rightarrow} \text{st emptys done (empty;}W\text{)}}
\end{array}
$$

The last thing to do is to apply the assumption *Subcomp*. This corresponds to applying the subcomputation lemma. Seven application of the L$\forall$ rule with the parameters $K, E, W, \overline{\text{emptys}}, \overline{\text{done}}, \overline{\text{empty}}, D$ yield the following result:

$$
\begin{array}{ll}
 & Subcomp, K \in \overline{\text{env}}, E \in \overline{\text{exp}}, W \in \overline{\text{val}}, D \in \overline{\text{feval } K\ E\ W} \\
 & U_7 \in \overline{\text{st (emptys;;}K\text{) (ev } F\text{\&done) empty} \overset{*}{\Rightarrow} \text{st emptys done (empty;}W\text{)}} \\
\vdash_\Sigma & U_7 \in \overline{\text{st (emptys;;}K\text{) (ev } E\text{\&done) empty} \overset{*}{\Rightarrow} \text{st emptys done (empty;}W\text{)}}
\end{array}
$$

id closes the branch. Obviously, everything works out under the assumption that the subcomputation lemma is proven. To obtain the actual proof, we have to cut this derivation with the deviation of the subcomputation lemma: The proof term of the subcomputation lemma has the form: $[P/Append](Q)$ (equation 5.1). Let R be the proof term of the equivalence theorem:

$$
\begin{array}{ll}
 & Subcomp \in \forall K : \text{env}.\ \forall E : \text{exp}.\ \forall W : \text{val}.\ \forall D : \text{feval } \underline{K}\ \underline{E}\ \underline{W}. \\
 & \forall H : \text{envstack}.\ \forall P : \text{program}.\ \forall S : \text{env}.\ \overline{\text{st (}\underline{H}\text{;;}\underline{K}\text{) (ev } \underline{E}\text{\&}\underline{P}\text{) } \underline{S} \overset{*}{\Rightarrow} \text{st } \underline{H}\ \underline{P}\ (\underline{S}\text{;}\underline{W}\text{)}} \\
\vdash_\Sigma & R \in \forall K : \text{env}.\ \forall E : \text{exp}.\ \forall W : \text{val}.\ \overline{\text{feval } K\ E\ W} \rightarrow \\
 & \overline{\text{st (emptys;;}K\text{) (ev } E\text{\&done) empty} \overset{*}{\Rightarrow} \text{st emptys done (empty;}W\text{)}}
\end{array}
$$

The application of the cut rule yields the following proof term for the equivalence theorem:

$$[[P/Append](Q)/Subcomp](R) \tag{5.2}$$

□

This completes the presentation of the language $\mathcal{T}$ and some of its meta theoretical results in MLF.

Chapter 6

Conclusion

In this thesis, we presented the meta logic MLF for the Horn fragment of LF. LF is well-suited to represent deductive systems. In section 2.1 we introduced as an example a toy programming language $\mathcal{T}$. $\mathcal{T}$ and the notions of operational and natural semantics were then represented in LF. Note, that the result of this representation remained in the Horn fragment of LF. Therefore the Horn fragment of LF is already powerful enough to represent non-trivial problems.

$\mathcal{T}$ showed also a further property of LF: Even meta theoretical results can be represented. The representation of the proof of the equivalence theorem showed how induction is transformed into LF objects and LF types. The implementation in Elf demonstrated, how the computational content of a proof can be accessed and used.

Currently, meta theoretical results are proven with pencil and paper. The proof of the subcomputation lemma 2.1 represents a typical meta theoretical proof. MLF is designed to support this proof work. The inference rule system of MLF is based on the sequent calculus for intuitionistic logic equipped with rules, to incorporate declarations from LF signatures into the proof process. In addition, it offers a general recursion rule which can be used to provide induction hypothesis and a case distinction rule. The case distinction rule is used in the proof for the subcomputation lemma. MLF also contains a cut rule: the cut rule allows the combination of already proven results. We showed in the example that if an external lemma is needed for the proof of a theorem, the proof proceeds in three steps. First, the external lemma is proven. Second, the theorem must be derived under the additional assumption that the external lemma holds. Third, both derivations are combined with the cut rule.

The purpose of MLF is it to keep a strict distinction between meta level and LF level. This is important, because MLF should be only an auxiliary device to reason about LF, but when the result is found, the objective is to transform everything back onto the LF level. The purity results guarantee, that there is a well-defined interface between MLF and LF.

It is clear — from a logical point of view — that applications of the cut rule are unwanted. The purity result for example holds only for the cut-free case. The question to ask is, whether the cut rule application is necessary or if it can be eliminated. We have shown the local reduction theorem as a first step towards a general cut-elimination theorem.

This thesis raises also a lot of questions for future research work. MLF has to be refined and implemented, and its theory has to be further developed:

1. Even though this thesis is very theoretical in its content, its objective is of practical nature: to develop an interactive proof assistant component for Elf. So far MLF has not been implemented yet. It is planned to write a prototype based on MLF as an extension of Elf.

2. From a theoretical point of view, the development of MLF is not yet finished: A major topic for future research will be the generalization of MLF as a meta logic for full LF. The inference rule system for MLF is very strict in the treatment of LF function types: No Π-type is allowed to occur in the right hand side of a sequent. But this is too restrictive. Consider the following assumption: $X \in \overline{\Pi x : \text{exp}.\text{exp}}$. It is worthwhile to examine if and how MLF could be generalized so that the following axiom application is allowed.

$$\frac{}{X \in \overline{\Pi x : \text{exp}.\text{exp}} \vdash_\Sigma X \in \overline{\Pi x : \text{exp}.\text{exp}}}\ \mathsf{id}'$$

   A very closely related question is, how higher order abstract syntax can be treated. If MLF is extended to full LF, the question is already answered.

3. Another very important research issue is if and how MLF proofs can be transformed into type families on the LF level. We proved purity results as a first step in this direction. We have seen in the example, that it is possible to represent the append lemma and the subcomputation lemma in LF. The problem is getting much more complicated if higher order abstract syntax is involved.

4. Some more work has to be done concerning the meta theory of MLF. It would be very nice to have a cut/case-elimination result. We believe, to obtain such a result much more theoretical work is necessary. The connection between MLF and LF must be studied in more detail.

5. The rule system of MLF contains the recursion rule. The recursion rule is defined with a side condition which enforces the proof term to be total. We did not give any details about this side condition. For a correct implementation of MLF, this judgement has to be defined appropriately.

We believe that MLF is a first promising step towards a meta reasoning component for LF. More work is still to be done.

Appendix A

MLF rules

A.1 Language of MLF

Object variable names:	x
Meta Variable names:	X
Object constant names:	c
Type constant names:	a

Formulae:	F	$::=$	$\forall X : A.F \mid \exists X : A.F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid 1 \mid \overline{A}$
Goal formulae:	G	$::=$	$\forall X : A_D.G \mid \exists X : A_G.G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \rightarrow G \mid 1 \mid \overline{A_G}$
Data formulae:	D	$::=$	$\forall X : A_G.D \mid \exists X : A_D.D \mid D_1 \wedge D_2 \mid D_1 \vee D_2 \mid G \rightarrow D \mid 1 \mid \overline{A_D}$
Core formulae:	C	$::=$	$\forall X : A_P.C \mid \exists X : A_P.C \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid C_1 \rightarrow C_2 \mid 1 \mid \overline{A_P}$
Program Patterns:	Q	$::=$	$(\texttt{unit}) \mid (\texttt{pair}\ X_1\ X_2) \mid (\texttt{inl}\ X) \mid (\texttt{inr}\ X) \mid (\texttt{inx}\ X_1\ X_2) \mid \overline{N}$
Programs:	P	$::=$	$X \mid (\texttt{unit}) \mid (\texttt{rec}\ X.P) \mid (\texttt{fun}\ X.P) \mid (\texttt{pair}\ P_1\ P_2) \mid (\texttt{inl}\ P)$
			$\mid (\texttt{inr}\ P) \mid (\texttt{inx}\ P_1\ P_2) \mid (\texttt{let}\ P_1\ \texttt{be}\ X\ \texttt{in}\ P_2) \mid (\texttt{app}\ P_1\ P_2) \mid \overline{M}$
			$\mid \begin{pmatrix} \texttt{case}\ P\ \texttt{of} \\ \quad Q^{(1)} \Rightarrow P^{(1)} \\ \quad \vdots \\ \mid \quad Q^{(n)} \Rightarrow P^{(n)} \end{pmatrix}$
Kinds:	K	$::=$	$\text{type} \mid \Pi x : A_G.\ K$
Types:	A	$::=$	$a \mid (A\ M) \mid \Pi x : A_1.\ A_2$
Atomic types:	A_P	$::=$	$a \mid (A_P\ M)$
Goal types:	A_G	$::=$	A_P
Data types:	A_D	$::=$	$A_P \mid \Pi x : A_G.\ A_D$
Objects:	M	$::=$	$\underline{P} \mid x \mid c \mid \lambda x : A_G.\ M \mid (M_1\ M_2)$
Pure Objects:	M	$::=$	$\underline{X} \mid x \mid c \mid \lambda x : A_G.\ M \mid (M_1\ M_2)$
Object Patterns:	N	$::=$	$c \mid (N\ \underline{X})$
Meta-context:	Γ	$::=$	$\cdot \mid \Gamma, X \in D$
Object-context:	Δ	$::=$	$\cdot \mid \Delta, x : A_D$
Signature:	Σ	$::=$	$\cdot \mid \Sigma, c : A_D \mid \Sigma, a : K$

A.2 Judgements for MLF

1.	$\vdash_\Sigma \Gamma$ ctx		
2.	$\Gamma \vdash_\Sigma G$ goal	$\Gamma \vdash_\Sigma D$ data	
3.	$\Gamma \vdash_\Sigma P \in G$		
4.	$\Gamma \vdash_\Sigma \Delta$ objctx		
5.	$\Gamma; \Delta \vdash_\Sigma K$ kind		
6.	$\Gamma; \Delta \vdash_\Sigma A_P : K$	$\Gamma; \Delta \vdash_\Sigma A_G : K$	$\Gamma; \Delta \vdash_\Sigma A_D : K$
7.	$\Gamma; \Delta \vdash_\Sigma M : A_P$	$\Gamma; \Delta \vdash_\Sigma M : A_G$	$\Gamma; \Delta \vdash_\Sigma M : A_D$
8.	$M_1 \equiv M_2$		
9.	$A_{P1} \equiv A_{P2}$	$A_{G1} \equiv A_{G2}$	$A_{D1} \equiv A_{D2}$
10.	$K_1 \equiv K_2$		

A.2.1 Typing rules for meta context

Judgment:

$$\vdash_\Sigma \Gamma \text{ ctx}$$

Rules:

$$\frac{}{\vdash_\Sigma \cdot \text{ ctx}}\ \mathsf{ctxemp} \qquad \frac{\vdash_\Sigma \Gamma \text{ ctx} \quad \Gamma \vdash_\Sigma D \text{ data}}{\vdash_\Sigma \Gamma, X \in D \text{ ctx}}\ \mathsf{ctxcons}$$

A.2.2 Typing rules for goals

Judgment:

$$\Gamma \vdash_\Sigma G \text{ goal}$$

Rules:

$$\frac{\Gamma; \cdot \vdash_\Sigma A_D : \text{type} \quad \Gamma, X \in \overline{A_D} \vdash_\Sigma G \text{ goal}}{\Gamma \vdash_\Sigma \forall X : A_D.G \text{ goal}}\ \mathsf{goalforall}$$

$$\frac{\Gamma; \cdot \vdash_\Sigma A_G : \text{type} \quad \Gamma, X \in \overline{A_G} \vdash_\Sigma G \text{ goal}}{\Gamma \vdash_\Sigma \exists X : A_G.G \text{ goal}}\ \mathsf{goalexists}$$

$$\frac{\Gamma \vdash_\Sigma G_1 \text{ goal} \quad \Gamma \vdash_\Sigma G_2 \text{ goal}}{\Gamma \vdash_\Sigma G_1 \wedge G_2 \text{ goal}}\ \mathsf{goaland}$$

$$\frac{\Gamma \vdash_\Sigma G_1 \text{ goal} \quad \Gamma \vdash_\Sigma G_2 \text{ goal}}{\Gamma \vdash_\Sigma G_1 \vee G_2 \text{ goal}} \textsf{goalor}$$

$$\frac{\Gamma \vdash_\Sigma D \text{ data} \quad \Gamma \vdash_\Sigma G \text{ goal}}{\Gamma \vdash_\Sigma D \to G \text{ goal}} \textsf{goalimp}$$

$$\frac{}{\Gamma \vdash_\Sigma 1 \text{ goal}} \textsf{goaltrue}$$

$$\frac{\Gamma; \cdot \vdash_\Sigma K \text{ kind} \quad \Gamma; \cdot \vdash_\Sigma A_G : K}{\Gamma \vdash_\Sigma \overline{A_G} \text{ goal}} \textsf{goaltype}$$

A.2.3 Typing rules for data formulae

Judgment:

$$\Gamma \vdash_\Sigma D \text{ data}$$

Rules:

$$\frac{\Gamma; \cdot \vdash_\Sigma A_G : \text{type} \quad \Gamma, X \in \overline{A_G} \vdash_\Sigma D \text{ data}}{\Gamma \vdash_\Sigma \forall X : A_G.D \text{ data}} \textsf{dataforall}$$

$$\frac{\Gamma; \cdot \vdash_\Sigma A_D : \text{type} \quad \Gamma, X \in \overline{A_D} \vdash_\Sigma D \text{ data}}{\Gamma \vdash_\Sigma \exists X : A_D.D \text{ data}} \textsf{dataexists}$$

$$\frac{\Gamma \vdash_\Sigma D_1 \text{ data} \quad \Gamma \vdash_\Sigma D_2 \text{ data}}{\Gamma \vdash_\Sigma D_1 \wedge D_2 \text{ data}} \textsf{dataand}$$

$$\frac{\Gamma \vdash_\Sigma D_1 \text{ data} \quad \Gamma \vdash_\Sigma D_2 \text{ data}}{\Gamma \vdash_\Sigma D_1 \vee D_2 \text{ data}} \textsf{dataor}$$

$$\frac{\Gamma \vdash_\Sigma G \text{ goal} \quad \Gamma \vdash_\Sigma D \text{ data}}{\Gamma \vdash_\Sigma G \to D \text{ data}} \textsf{dataimp}$$

$$\frac{}{\Gamma \vdash_\Sigma 1 \text{ data}} \textsf{datatrue}$$

$$\frac{\Gamma; \cdot \vdash_\Sigma K \text{ kind} \quad \Gamma; \cdot \vdash_\Sigma A_D : K}{\Gamma \vdash_\Sigma \overline{A_D} \text{ data}} \textsf{datatype}$$

A.2.4 Typing rules for programs

Judgment:

$$\Gamma \vdash_\Sigma P \in G$$

Rules:

$$\frac{\vdash_\Sigma \Gamma_1, X \in C, \Gamma_2 \ \mathrm{ctx}}{\Gamma_1, X \in C, \Gamma_2 \vdash_\Sigma X \in C}\,\mathsf{id}$$

$$\frac{\vdash_\Sigma \Gamma \ \mathrm{ctx}}{\Gamma \vdash_\Sigma \overline{c} \in \overline{A_G}}\,\mathsf{const} \qquad \text{for } c : A_G \text{ defined in } \Sigma$$

$$\frac{\vdash_\Sigma \Gamma \ \mathrm{ctx}}{\Gamma \vdash_\Sigma (\mathtt{unit}) \in 1}\,\mathsf{R1}$$

$$\frac{\Gamma \vdash_\Sigma P_1 \in G_1 \quad \Gamma \vdash_\Sigma P_2 \in G_2}{\Gamma \vdash_\Sigma (\mathtt{pair}\ P_1\ P_2) \in G_1 \wedge G_2}\,\mathsf{R}\wedge$$

$$\frac{\Gamma \vdash_\Sigma P \in G_1}{\Gamma \vdash_\Sigma (\mathtt{inl}\ P) \in G_1 \vee G_2}\,\mathsf{R}\vee_1 \qquad \frac{\Gamma \vdash_\Sigma P \in G_2}{\Gamma \vdash_\Sigma (\mathtt{inr}\ P) \in G_1 \vee G_2}\,\mathsf{R}\vee_2$$

$$\frac{\Gamma, X \in D \vdash_\Sigma P \in G}{\Gamma \vdash_\Sigma (\mathtt{fun}\ X.P) \in D \to G}\,\mathsf{R} \to$$

$$\frac{\Gamma, Y \in \overline{A_D} \vdash_\Sigma [Y/X]P \in [Y/X](G)}{\Gamma \vdash_\Sigma (\mathtt{fun}\ X.P) \in \forall X : A_D.G}\,\mathsf{R}\forall$$

$$\frac{\Gamma \vdash_\Sigma P' \in \overline{A_G} \quad \Gamma \vdash_\Sigma P \in [P'/X](G)}{\Gamma \vdash_\Sigma (\mathtt{inx}\ P'\ P) \ \in \exists X : A_G.G}\,\mathsf{R}\exists$$

$$\frac{\Gamma, X \in C \vdash_\Sigma P \in C}{\Gamma \vdash_\Sigma (\mathtt{rec}\ X.P) \in C}\,\mathsf{rec} \quad \text{with } P \downarrow X$$

$$\frac{\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G}{\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2 \vdash_\Sigma (\mathtt{case}\ X\ \mathtt{of}\ (\mathtt{pair}\ X_1\ X_2) \Rightarrow P) \in G}\,\mathsf{L}\wedge$$

$$\frac{\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, X_1 \in D_1 \vdash_\Sigma P_1 \in G \quad \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, X_2 \in D_2 \vdash_\Sigma P_2 \in G}{\Gamma_1, X \in D_1 \vee D_2, \Gamma_2 \vdash_\Sigma \left(\begin{array}{ll} \mathtt{case}\ X\ \mathtt{of} & (\mathtt{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\mathtt{inr}\ X_2) \Rightarrow P_2 \end{array} \right) \in G}\ \mathsf{L}\vee$$

$$\frac{\Gamma_1, X \in G_1 \to D, \Gamma_2 \vdash_\Sigma P_1 \in G_1 \quad \Gamma_1, X \in G_1 \to D, \Gamma_2, Y \in D \vdash_\Sigma P_2 \in G_2}{\Gamma_1, X \in G_1 \to D, \Gamma_2 \vdash_\Sigma (\mathtt{let}\ (\mathtt{app}\ X\ P_1)\ \mathtt{be}\ Y\ \mathtt{in}\ P_2) \in G_2}\ \mathsf{L}\to$$

$$\frac{\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma P_1 \in \overline{A_G} \quad \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, Z \in [P_1/Y](D) \vdash_\Sigma P_2 \in G}{\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2 \vdash_\Sigma (\mathtt{let}\ (\mathtt{app}\ X\ P_1)\ \mathtt{be}\ Z\ \mathtt{in}\ P_2) \in G}\ \mathsf{L}\forall$$

$$\frac{\Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D) \vdash_\Sigma P \in G}{\Gamma_1, X \in \exists Y : A_D.D, \Gamma_2 \vdash_\Sigma (\mathtt{case}\ X\ \mathtt{of}\ (\mathtt{inx}\ X_1\ X_2) \Rightarrow P) \in G}\ \mathsf{L}\exists$$

$$\frac{\begin{array}{l} \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2 \vdash_\Sigma \overline{M} \in \overline{A_G} \\ \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, Y \in \overline{\{M/x\}_{\mathrm{type}}(A_D)} \vdash_\Sigma P \in G \end{array}}{\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2 \vdash_\Sigma [\overline{\underline{X}\ M}/Y](P) \in G}\ \mathsf{L}\Pi$$

$$\frac{\begin{array}{l} \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2 \vdash_\Sigma Z \in \overline{A_G} \\ \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, Y \in \overline{\{\underline{Z}/x\}_{\mathrm{type}}(A_D)} \vdash_\Sigma P \in G \end{array}}{\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2 \vdash_\Sigma [\overline{\underline{X}\ \underline{Z}}/Y](P) \in G}\ \mathsf{L}\Pi\mathsf{V}$$

$$\frac{\Gamma \vdash_\Sigma \overline{M} \in \overline{A_G} \quad \Gamma, Y \in \overline{\{M/x\}_{\mathrm{type}}(A_D)} \vdash_\Sigma P \in G}{\Gamma \vdash_\Sigma [\overline{c\ M}/Y](P) \in G}\ \mathsf{L}\Pi\Sigma \quad \text{where } c : \Pi x : A_G.A_D \in \Sigma$$

$$\frac{\Gamma \vdash_\Sigma Z \in \overline{A_G} \quad \Gamma, Y \in \overline{\{\underline{Z}/x\}_{\mathrm{type}}(A_D)} \vdash_\Sigma P \in G}{\Gamma \vdash_\Sigma [\overline{c\ \underline{Z}}/Y](P) \in G}\ \mathsf{L}\Pi\Sigma\mathsf{V} \quad \text{where } c : \Pi x : A_G.A_D \in \Sigma$$

$$\frac{\vdash_\Sigma [P/X](\Gamma)\ \mathrm{ctx} \quad \Gamma \vdash_\Sigma P \in \overline{A_G} \quad \overset{\text{for all } i \le n}{\Lambda^{(i)}, [\Theta^{(i)}](\Gamma') \vdash_\Sigma P^{(i)} \in [\Theta^{(i)}](G')}}{[P/X](\Gamma) \vdash_\Sigma \left(\begin{array}{ll} \mathtt{case}\ P\ \mathtt{of} & \\ & \overline{c_1\ \underline{Y_1^{(1)}} ... \underline{Y_{m_1}^{(1)}}} \Rightarrow [\eta](P^{(1)}) \\ & ... \\ \mid & \overline{c_n\ \underline{Y_1^{(n)}} ... \underline{Y_{m_n}^{(n)}}} \Rightarrow [\eta](P^{(n)}) \end{array} \right) \in [P/X](G)}\ \mathsf{case}$$

where following side conditions hold:

1. $\mathrm{Ind}_{\Sigma,\Gamma}(X, A_G') = \{\langle \Lambda^{(1)}, \Theta^{(1)} \rangle .. \langle \Lambda^{(n)}, \Theta^{(n)} \rangle\}$
2. There is a η s.t. $[\eta](A_G') = A_G$, $[\eta](\Gamma') = \Gamma$ and $[\eta](G') = G$

$$\frac{\Gamma_1 \vdash_\Sigma P \in C \quad \Gamma_1, X \in C, \Gamma_2 \vdash_\Sigma P' \in G}{\Gamma_1, [P/X](\Gamma_2) \vdash_\Sigma [P/X](P') \in [P/X](G)}\ \mathsf{cut}$$

A.2.5 Typing rules for object context

Judgment:

$$\Gamma \vdash_\Sigma \Delta \text{ objctx}$$

Rules:

$$\frac{}{\Gamma \vdash_\Sigma \cdot \text{ objctx}}\ \textsf{objctxemp}$$

$$\frac{\Gamma \vdash_\Sigma \Delta \text{ objctx} \quad \Gamma; \Delta \vdash_\Sigma A_D : \text{type}}{\Gamma \vdash_\Sigma \Delta, x : A_D \text{ objctx}}\ \textsf{objctxcons}$$

A.2.6 Typing rules for kinds

Judgment:

$$\Gamma; \Delta \vdash_\Sigma K \text{ kind}$$

Rules:

$$\frac{}{\Gamma; \Delta \vdash_\Sigma \text{type kind}}\ \textsf{kindtype}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma A_G : \text{type} \quad \Gamma; \Delta, x : A_G \vdash_\Sigma K \text{ kind}}{\Gamma; \Delta \vdash_\Sigma \Pi x : A_G.\ K \text{ kind}}\ \textsf{kindpi}$$

Note, that every A_G is also an A_D.

A.2.7 Typing rules for atomic types

Judgment:

$$\Gamma; \Delta \vdash_\Sigma A_P : K$$

Rules:

$$\frac{\Sigma(a) = K}{\Gamma; \Delta \vdash_\Sigma a : K}\ \textsf{typeatomconst}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma A_P : \Pi x : A_G.\ K \quad \Gamma; \Delta \vdash_\Sigma M : A_G}{\Gamma; \Delta \vdash_\Sigma (A_P\, M) : \{M/x\}_{\text{kind}}(K)}\ \textsf{typeatomapp}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma A_P : K \quad K \equiv K' \quad \Gamma; \Delta \vdash_\Sigma K' : \text{kind}}{\Gamma; \Delta \vdash_\Sigma A_P : K'}\ \textsf{typeatomequiv}$$

A.2.8 Typing rules for goal types

Judgment:

$$\Gamma; \Delta \vdash_\Sigma A_G : K$$

Rules:

No new rules

A.2.9 Typing rules for data types

Judgment:

$$\Gamma; \Delta \vdash_\Sigma A_D : K$$

Rules:

$$\frac{\Gamma; \Delta \vdash_\Sigma A_G : \text{type} \quad \Gamma; \Delta, x : A_G \vdash_\Sigma A_D : \text{type}}{\Gamma; \Delta \vdash_\Sigma \Pi x : A_G .\, A_D : \text{type}}\ \mathsf{typedatapi}$$

A.2.10 Typing rules for objects of atomic type

Judgment:

$$\Gamma; \Delta \vdash_\Sigma M : A_P$$

Rules:

no rules

A.2.11 Typing rules for objects of goal type

Judgment:

$$\Gamma; \Delta \vdash_\Sigma M : A_G$$

Rules:
Impure MLF:

$$\frac{\Gamma \vdash_\Sigma P \in \overline{A_G}}{\Gamma; \Delta \vdash_\Sigma \underline{P} : A_G}\ \mathsf{objgoalprgI}$$

Pure MLF:

$$\frac{\Gamma(X) = \overline{A_G}}{\Gamma; \Delta \vdash_\Sigma \underline{X} : A_G}\ \mathsf{objgoalprgP}$$

A.2.12 Typing rules for objects of data type

Judgment:

$$\Gamma; \Delta \vdash_\Sigma M : A_D$$

Rules:

$$\frac{\Delta(x) = A_D}{\Gamma; \Delta \vdash_\Sigma x : A_D}\ \textsf{objdatasigma}$$

$$\frac{\Sigma(c) = A_D}{\Gamma; \Delta \vdash_\Sigma c : A_D}\ \textsf{objdataconst}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma M_1 : \Pi x : A_G.\, A_D \quad \Gamma; \Delta \vdash_\Sigma M_2 : A_G}{\Gamma; \Delta \vdash_\Sigma (M_1\, M_2) : A_D}\ \textsf{objdataapp}$$

$$\frac{\Gamma; \Delta, x : A_G \vdash_\Sigma M : A_D}{\Gamma; \Delta \vdash_\Sigma \lambda x : A_G.\, M : \Pi x : A_G.\, A_D}\ \textsf{objdatapi}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma M : A_D \quad A_D \equiv A_D{}' \quad \Gamma; \Delta \vdash_\Sigma A_D{}' : \text{type}}{\Gamma; \Delta \vdash_\Sigma M : A_D{}'}\ \textsf{objdataequiv}$$

no rules for typing programs

A.2.13 Congruence relation for kinds

Judgment:

$$K_1 \equiv K_2$$

Rules:

same as in LF [Pfe92, HHP93].

A.2.14 Congruence relation for atomic types

Judgment:

$$A_{P1} \equiv A_{P2}$$

Rules:

transitivity and congruence as in LF.

A.2.15 Congruence relation for goal types

Judgment:

$$A_{G1} \equiv A_{G2}$$

Rules:

similar to Subsection A.2.14.

A.2.16 Congruence relation for data types

Judgment:

$$A_{D1} \equiv A_{D2}$$

Rules:

transitivity and congruence rules as in LF.

A.2.17 Congruence relation for objects

Judgment:

$$M_1 \equiv M_2$$

Rules:

$$\frac{}{(\lambda x : A_G.\ M)N \equiv \{N/x\}_{\mathsf{object}}(M)}\ \mathsf{objbeta}$$

$$\frac{}{(\lambda x : A_G.\ (M\ x)) \equiv M}\ \mathsf{objeta}$$

Impure MLF:

$$\frac{}{\underline{P} \equiv \underline{P}}\ \mathsf{objprgI}$$

Pure MLF:

$$\frac{}{\underline{X} \equiv \underline{X}}\ \mathsf{objprgP}$$

transitivity and congruence as in LF.

Appendix B

Purity proofs

Lemma 4.1 *Let M' be a pure object, A' a pure type, and Θ a strictly pure substitution, i.e. $\Theta(X) = Y$ or $\Theta(X) = \overline{M}$ for all $X \in dom(\Theta)$. Then $\Theta(M')$ is pure and $\Theta(A')$ is pure.*

Proof: by mutual induction over the structure of A', M'.

Case: $M' = \underline{X}$. If $\Theta(X) = Y$ then

$$[\Theta]_{\text{object}}(M') = [\Theta]_{\text{object}}(\underline{X}) = \underline{[\Theta]_{\text{program}}(X)} = \underline{Y}$$

which is pure. Note, that $X = Y$ is possible. In the other case, if $\Theta(X) = \overline{M}$, then

$$[\Theta]_{\text{object}}(M') = [\Theta]_{\text{object}}(\underline{X}) = M$$

is pure.

Case: $M' = x$.

$$[\Theta]_{\text{object}}(M') = [\Theta]_{\text{object}}(x) = x$$

is pure.

Case: $M' = c$.

$$[\Theta]_{\text{object}}(M') = [\Theta]_{\text{object}}(c) = c$$

is pure.

Case: $M' = \lambda x : A_G{}'.\, M''$.

$$[\Theta]_{\text{object}}(M') = [\Theta]_{\text{object}}(\lambda x : A_G{}'.\, M'') = \lambda x : [\Theta]_{\text{type}}(A_G{}').\, [\Theta]_{\text{object}}(M'')$$

Induction hypothesis gives us that $[\Theta]_{\text{type}}(A_G{}')$ and $[\Theta]_{\text{object}}(M'')$ are pure. Therefore $[\Theta]_{\text{object}}(M')$ is pure.

Case: $M' = (M_1' \; M_2')$.

$$[\Theta]_{\text{object}}(M') = [\Theta]_{\text{object}}(M_1' \; M_2') = ([\Theta]_{\text{object}}(M_1') \; [\Theta]_{\text{object}}(M_2'))$$

Induction hypothesis gives us $[\Theta]_{\text{object}}(M_1'), [\Theta]_{\text{object}}(M_2')$ are pure. Therefore $[\Theta]_{\text{object}}(M')$ is pure.

Case: $A' = a$:

$$[\Theta]_{\text{type}}(A') = [\Theta]_{\text{type}}(a) = a$$

is pure.

Case: $A' = \Pi x : A'_1.\ A'_2$.

$$[\Theta]_{\text{type}}(A') = [\Theta]_{\text{type}}(\Pi x : A'_1.\ A'_2) = \Pi x : [\Theta]_{\text{type}}(A'_1).\ [\Theta]_{\text{type}}(A'_2)$$

Induction hypothesis gives us $[\Theta]_{\text{type}}(A'_1), [\Theta]_{\text{type}}(A'_2)$ are pure. Therefore $[\Theta]_{\text{type}}(A')$ is pure.

Case: $A' = (A'_1\ M'_1)$.

$$[\Theta]_{\text{type}}(A') = [\Theta]_{\text{type}}(A'_1\ M'_1) = ([\Theta]_{\text{type}}(A'_1)\ [\Theta]_{\text{object}}(M'_1))$$

Induction hypothesis gives us $[\Theta]_{\text{type}}(A'_1), [\Theta]_{\text{object}}(M'_1)$ are pure. Therefore $[\Theta]_{\text{object}}(M')$ is pure.

□

Lemma 4.2 *Let Θ be a substitution and $[\Theta](M)$ be a pure object. Then M is pure.*

Proof: Assume the contrary. M is not pure implies $[\Theta](M)$ not pure: There is a subobject M_1 of the form $\underline{P}$, and $P \neq X$ a variable name.

$$[\Theta](M_1) = [\Theta](\underline{P}) = \underline{[\Theta](P)} \neq \underline{[\Theta](Y)}$$

Since $[\Theta](M_1)$ is a subobject of $[\Theta](M)$, $[\Theta](M)$ cannot be pure. □

Lemma 4.3 *Let Θ be a substitution and $[\Theta](A)$ be a pure type. Then A is pure.*

Proof: Assume the contrary. A is not pure implies $[\Theta](A)$ not pure: There is a syntactical subtype of the form $(A_1\ M)$, with M not pure. By lemma 4.2 we obtain $[\Theta](M)$ not pure. Therefore $[\Theta](A_1\ M)$ is not pure, and since this is a syntactical subtype of $[\Theta](A)$, it cannot be pure. □

Lemma 4.4 *Let Θ be substitution, M object and $[\Theta]_{object}(M)$ pure and A be a type and $[\Theta]_{type}(A)$ pure. Then $\Theta|_{Free(M)}$ must be strict and $\Theta|_{Free(A)}$ is strict.*

Proof: By lemma 4.2 we have M is pure. Proof by mutual induction over the structure of M is pure and A is pure.

Case: $M = \underline{X}$ then

$$[\Theta]_{\text{object}}(M) = [\Theta]_{\text{object}}(\underline{X}) = M'$$

where $M' \in \{\underline{Y}, c, \lambda x : A_G''.\ M'', (M_1''\ M_2'')\}$ which must be the domain of Θ, therefore $\Theta|_{\{X\}}$ is strictly pure.

Case: $M = x$.

$$[\Theta]_{\text{object}}(M) = x$$

Θ doesn't do anything in this case, therefore, $\Theta|_\emptyset$ is strictly pure.

Case: $M = c$.

$$[\Theta]_{\text{object}}(M) = c$$

Therefore $\Theta|_\emptyset$ is strictly pure.

Case: $M = \lambda x : A_G.\, M_1$.

$$[\Theta]_{\text{object}}(M) = [\Theta]_{\text{object}}(\lambda x : A_G.\, M_1) = \lambda x : [\Theta]_{\text{type}}(A_G).\, [\Theta]_{\text{object}}(M_1)$$

A_G pure type and $[\Theta]_{\text{type}}(A_G)$ pure gives us $\Theta|_{Free(A_G)}$ is strict. M_1 pure type and $[\Theta]_{\text{object}}(M_1)$ pure gives us $\Theta|_{Free(M_1)}$ is strict. Therefore $\Theta|_{Free(M)} = \Theta|_{Free(A_G) \cup Free(M_1)} = \Theta|_{Free(A_G)} \cup \Theta|_{Free(M_2)}$ is strict.

Case: $M = (M_1\ M_2)$.

$$[\Theta]_{\text{object}}(M) = [\Theta]_{\text{object}}(M_1\ M_2) = ([\Theta]_{\text{object}}(M_1)\ [\Theta]_{\text{object}}(M_2))$$

M_1 pure type and $[\Theta]_{\text{object}}(M_1)$ pure gives us $\Theta|_{Free(M_1)}$ is strict. M_2 pure type and $[\Theta]_{\text{object}}(M_2)$ pure gives us $\Theta|_{Free(M_2)}$ is strict. Therefore $\Theta|_{Free(M)} = \Theta|_{Free(M_1) \cup Free(M_2)} = \Theta|_{Free(M_1)} \cup \Theta|_{Free(M_2)}$ is strict.

Case: $A = a$:

$$[\Theta]_{\text{object}}(A) = [\Theta]_{\text{object}}(a) = a$$

$\Theta|_\emptyset$ is strictly pure.

Case: $A = \Pi x : A_1.\, A_2$.

$$[\Theta]_{\text{type}}(A) = [\Theta]_{\text{type}}(\Pi x : A_1.\, A_2) = \Pi x : [\Theta]_{\text{type}}(A_1).\, [\Theta]_{\text{type}}(A_2)$$

A_1 pure type and $[\Theta]_{\text{type}}(A_1)$ pure gives us $\Theta|_{Free(A_1)}$ is strict. A_2 pure type and $[\Theta]_{\text{type}}(A_2)$ pure gives us $\Theta|_{Free(A_2)}$ is strict. Therefore $\Theta|_{Free(A)} = \Theta|_{Free(A_1) \cup Free(A_2)} = \Theta|_{Free(A_1)} \cup \Theta|_{Free(A_2)}$ is strict.

Case: $A = (A_1\ M_1)$.

$$[\Theta]_{\text{type}}(A) = [\Theta]_{\text{type}}(A_1\ M_1) = ([\Theta]_{\text{type}}(A_1)\ [\Theta]_{\text{object}}(M_1))$$

A_1 pure type and $[\Theta]_{\text{type}}(A_1)$ pure gives us $\Theta|_{Free(A_1)}$ is strict. M_1 pure type and $[\Theta]_{\text{object}}(M_1)$ pure gives us $\Theta|_{Free(M_1)}$ is strict. Therefore $\Theta|_{Free(A)} = \Theta|_{Free(A_1) \cup Free(M_1)} = \Theta|_{Free(A_1)} \cup \Theta|_{Free(M_1)}$ is strict.

$\square$

Lemma 4.5 *Let P' be a pure program and Θ a strictly pure substitution. Then $\Theta(P')$ is pure.*

Proof: by induction over the structure of P'.

Case: $P' = X$. If $[\Theta]_{\text{program}}(X) = Y$ then

$$[\Theta]_{\text{program}}(P') = [\Theta]_{\text{program}}(X) = Y$$

which is pure, else if $[\Theta]_{\text{program}}(X) = \overline{M'}$ then

$$[\Theta]_{\text{program}}(P') = [\Theta]_{\text{program}}(\overline{M'}) = \overline{[\Theta]_{\text{object}}(M')}$$

which is pure because of lemma 4.1 $[\Theta]_{\text{object}}(M')$ is pure. Else

$$[\Theta]_{\text{program}}(P') = [\Theta]_{\text{program}}(X) = X$$

which is pure by definition.

Case: $P' = (\texttt{unit})$ is pure by definition.

Case: $P' = (\texttt{rec}\ X.P'')$.

$$[\Theta]_{\text{program}}(P') = [\Theta]_{\text{program}}(\texttt{rec}\ X.P'') = (\texttt{rec}\ X.[\Theta]_{\text{program}}(P''))$$

is pure because $[\Theta]_{\text{program}}(P'')$ is pure by induction hypothesis.

Case: $P' = (\texttt{fun}\ X.P'')$ analog.

Case: $P' = (\texttt{pair}\ P_1'\ P_2')$ analog.

Case: $P' = (\texttt{inl}\ P'')$ analog.

Case: $P' = (\texttt{inr}\ P'')$ analog.

Case: $P' = (\texttt{inx}\ P_1'\ P_2')$ analog.

Case: $P' = (\texttt{app}\ P_1'\ P_2')$ analog.

Case: $P' = (\texttt{let}\ P_1'\ \texttt{be}\ X\ \texttt{in}\ P_2')$.

$$[\Theta]_{\text{program}}(P') = [\Theta]_{\text{program}}(\texttt{let}\ P_1'\ \texttt{be}\ X\ \texttt{in}\ P_2') = (\texttt{let}\ [\Theta]_{\text{program}}(P_1')\ \texttt{be}\ Y\ \texttt{in}\ [\Theta, Y/X]_{\text{program}}(P_2'))$$

Because of induction hypothesis $[\Theta]_{\text{program}}(P_1')$ is pure, and since $\Theta, Y/X$ is also strict, $[\Theta, Y/X]_{\text{program}}(P_2')$ is pure by induction hypothesis.

Case: $P' = \overline{M'}$.

$$[\Theta]_{\text{program}}(P') = [\Theta]_{\text{program}}(\overline{M'}) = \overline{[\Theta]_{\text{object}}(M')}$$

which is pure because $[\Theta]_{\text{object}}(M')$ is pure by induction hypothesis.

Case: $P' = \left(\begin{array}{l} \texttt{case}\ P\ \texttt{of} \\ \quad Q^{(1)} \Rightarrow P^{(1)} \\ \quad \vdots \\ \mid \quad Q^{(n)} \Rightarrow P^{(n)} \end{array} \right)$

$$[\Theta]_{\text{program}}(P') = [\Theta]_{\text{program}}(\left(\begin{array}{l} \texttt{case}\ P\ \texttt{of} \\ \quad Q^{(1)} \Rightarrow P(1) \\ \quad \vdots \\ \mid \quad Q^{(n)} \Rightarrow P^{(n)} \end{array} \right))$$

$$= \left(\begin{array}{l} \texttt{case}\ [\Theta]_{\text{program}}(P)\ \texttt{of} \\ \quad [\Psi_1]_{\text{program}}(Q^{(1)}) \Rightarrow [\Theta \circ \Psi_1]_{\text{program}}(P^{(1)}) \\ \quad \vdots \\ \mid \quad [\Psi_1]_{\text{program}}(Q^{(n)}) \Rightarrow [\Theta \circ \Psi_n]_{\text{program}}(P^{(n)}) \end{array} \right)$$

is pure, because $[\Theta]_{\text{program}}(P)$ is pure, $\Theta \circ \Psi_k$ is strictly pure — this is because Ψ_k is only a variable renaming substitution — and $[\Theta \circ \Psi_k]_{\text{program}}(P^{(k)})$ are pure for $k \leq n$ by induction hypothesis.

□

Lemma 4.6 *Let Θ be a substitution and $[\Theta](P)$ be a pure. Then P is pure.*

Proof: Assume the contrary. P is not pure implies $[\Theta](P)$ not pure: There is a syntactical subprogram of the form $\overline{M}$, with M not pure. By lemma 4.2 we obtain $[\Theta](M)$ not pure. Therefore $\overline{[\Theta](M)}$ is not pure, and since this is a syntactical subprogram of $[\Theta](P)$, it cannot be pure. □

Lemma 4.7 *Let G' be a pure formula and Θ a strictly pure substitution. Then $\Theta(G')$ is pure.*

Proof: by induction over the structure of G'.

Case: $G' = \forall X : A'.G''$:

$$[\Theta]_{\text{formula}}(G') = [\Theta]_{\text{formula}}(\forall X : A'.G'') = \forall Y : [\Theta]_{\text{type}}(A)[\Theta \circ (Y/X)]_{\text{formula}}(G'')$$

is pure because $[\Theta]_{\text{type}}(A)$ is pure due to lemma 4.1, $\Theta \circ (Y/X)$ is a strictly pure substitution and $[\Theta \circ (Y/X)]_{\text{formula}}(G'')$ is pure by induction hypothesis.

Case: $G' = \exists X : A'.G''$: analog

Case: $G' = G'_1 \wedge G'_2$:

$$[\Theta]_{\text{formula}}(G') = [\Theta]_{\text{formula}}(G'_1 \wedge G'_2) = [\Theta]_{\text{formula}}(G'_1) \wedge [\Theta]_{\text{formula}}(G'_2)$$

is pure because $[\Theta]_{\text{formula}}(G'_1)$, $[\Theta]_{\text{formula}}(G'_2)$ are pure by induction hypothesis.

Case: $G' = G'_1 \vee G'_2$: analog.

Case: $G' = G'_1 \to G'_2$: analog.

Case: $G' = 1$ is pure by definition.

Case: $G' = \overline{A'}$:

$$[\Theta]_{\text{formula}}(G') = [\Theta]_{\text{formula}}(\overline{A'}) = \overline{[\Theta]_{\text{formula}}(A')}$$

is pure because $[\Theta]_{\text{formula}}(A')$ is pure by lemma 4.1.

□

Lemma 4.8 *Let Θ be a substitution and $[\Theta](G)$ be a pure formula. Then G is pure.*

Proof: Assume the contrary. G is not pure implies $[\Theta](G)$ not pure: There is a syntactical subprogram of G of the form $\overline{A}$, with A not pure. By lemma 4.3 we obtain $[\Theta](A)$ not pure. Therefore $\overline{[\Theta](A)}$ is not pure, and since this is a syntactical subprogram of $[\Theta](G)$, it cannot be pure. □

Lemma 4.9 *Let Θ be substitution, G formula and $[\Theta]_{formula}(G)$ pure. Then $\Theta|_{Free(G)}$ is strict.*

Proof: By lemma 4.8 we have G is pure. Proof by mutual induction over the structure of G is pure.

Case: $G = \forall X : A_1.G_1$:

$$[\Theta]_{\text{formula}}(G) = [\Theta]_{\text{formula}}(\forall X : A_1.G_1) = \forall Y : [\Theta]_{\text{type}}(A_1)[\Theta \circ (Y/X)]_{\text{formula}}(G_1)$$

A_1 pure type and $[\Theta]_{\text{type}}(A_1)$ pure gives us $\Theta|_{Free(A_1)}$ is strict by lemma 4.4. G_1 pure formula, therefore $[Y/X]_{\text{formula}}(G_1)$ pure formula. $[\Theta \circ (Y/X)]_{\text{formula}}(G_1)$ pure implies $[\Theta]_{\text{formula}}([Y/X]_{\text{formula}}(G_1))$ gives us $\Theta|_{Free([Y/X]_{\text{formula}}(G_1))}$ is strict by induction hypothesis. Therefore $\Theta|_{Free(G)} = \Theta|_{Free(A_1) \cup Free(G_1) \setminus \{X\}} = \Theta|_{Free(A_1)} \cup \Theta|_{Free(G_1) \setminus \{X\}} = \Theta|_{Free(A_1)} \cup \Theta|_{Free([Y/X]_{\text{formula}}(G_1))}$ is strict.

Case: $G = \exists X : A_1.G_1$: analog

Case: $G = G_1 \wedge G_2$:

$$[\Theta]_{\text{formula}}(G) = [\Theta]_{\text{formula}}(G_1 \wedge G_2) = [\Theta]_{\text{formula}}(G_1) \wedge [\Theta]_{\text{formula}}(G_2)$$

G_1 pure formula and $[\Theta]_{\text{type}}(G_1)$ pure gives us $\Theta|_{Free(G_1)}$ is strict by induction hypothesis. G_2 pure formula and $[\Theta]_{\text{type}}(G_2)$ pure gives us $\Theta|_{Free(G_2)}$ is strict by induction hypothesis. Therefore $\Theta|_{Free(G)} = \Theta|_{Free(G_1) \cup Free(G_2)} = \Theta|_{Free(G_1)} \cup \Theta|_{Free(G_2)}$ is strict.

Case: $G = G_1 \vee G_2$: analog

Case: $G = G_1 \to G_2$: analog

Case: $G = 1$ is pure by definition.

Case: $G = \overline{A}$:

$$[\Theta]_{\mathrm{formula}}(G) = [\Theta]_{\mathrm{formula}}(\overline{A}) = \overline{[\Theta]_{\mathrm{type}}(A)}$$

A pure type and $[\Theta]_{\mathrm{type}}(A)$ pure gives us $\Theta|_{Free(A)}$ is strict by lemma 4.4. Therefore $\Theta|_{Free(G)} = \Theta|_{Free(A)}$ is strict.

□

Lemma 4.10 *Let Γ' be a pure context and Θ a strictly pure substitution. Then $\Theta(\Gamma')$ is pure.*

Proof: by induction over the structure of Γ.

Case: $\Gamma' = \cdot$ is pure by definition

Case: $\Gamma' = \Gamma'', X \in D'$ If $X \in dom(\Theta)$, we get

$$[\Theta]_{\mathrm{context}}(\Gamma') = [\Theta]_{\mathrm{context}}(\Gamma'', X \in D') = [\Theta]_{\mathrm{context}}(\Gamma'')$$

is pure because of induction hypothesis. If $X \notin dom(\Theta)$, we get

$$[\Theta]_{\mathrm{context}}(\Gamma') = [\Theta]_{\mathrm{context}}(\Gamma'', X \in D') = [\Theta]_{\mathrm{context}}(\Gamma''), X \in [\Theta]_{\mathrm{formula}}(D')$$

which is pure because $[\Theta]_{\mathrm{context}}(\Gamma'')$ because of induction hypothesis and $[\Theta]_{\mathrm{formula}}(D')$ is pure because of lemma 4.7.

□

Lemma 4.11 *Let Θ be a substitution and $[\Theta](\Gamma)$ be a pure context. Then Γ is pure.*

Proof: Assume the contrary. Γ is not pure implies $[\Theta](\Gamma)$ not pure: There is a syntactical subcontext of Γ of the form $\Gamma_1, X \in D$, with D not pure. By lemma 4.8 we obtain $[\Theta](D)$ not pure. Therefore $[\Theta](\Gamma_1, X \in D)$ is not pure, and since this is a syntactical subcontext of $[\Theta](\Gamma)$, cannot be pure. □

Theorem 4.12 *Every typing rule in MLF without cut is purity preserving. That is, when the premisses are pure (all participating objects, types, programs, formulae, and contexts are pure) the conclusion will be pure, too.*

Proof: Case: id preserves purity: Under the assumption that $\Gamma_1, X \in C, \Gamma_2$ is pure, it follows trivially that X is pure and C is pure.

Case: const preserves purity because Γ is assumed to be pure.

Case: R1: analog.

Case: R$\wedge$: Γ pure, P_1, P_2 pure, and G_1, G_2 pure implies $(\mathtt{pair}\ P_1\ P_2), G_1 \wedge G_2$ pure.

Case: R$\vee_1$: analog

Case: R$\vee_2$: analog

Case: $\mathsf{R} \rightarrow$: analog

Case: $\mathsf{R}\forall$: Under the assumption that Γ is pure, and A_D is pure, and because of lemma 4.6 and lemma 4.8 we can conclude that P and G are pure, and therefore $(\texttt{fun}\ X.P)$ and $\forall X : A_D.G$ are pure.

Case: $\mathsf{R}\exists$: Under the assumption that Γ is pure, P, P' are pure, $[P'/X](G)$ is pure and A_G pure, we obtain by lemma 4.8 that G is pure. Consequently $(\texttt{inx}\ P'\ P)$ and $\exists X : A_G.G$ are pure.

Case: rec: analog

Case: $\mathsf{L}\wedge$: analog

Case: $\mathsf{L}\vee$: analog

Case: $\mathsf{L} \rightarrow$: analog

Case: $\mathsf{L}\forall$: analog

Case: $\mathsf{L}\exists$: analog

Case: $\mathsf{L}\Pi$: Assume $\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2$ is pure, $\overline{M}$ is pure and $\overline{A_G}$ is pure. From the second premiss we can assume additionally that P' is pure. Hence, since $[\underline{X}\ \overline{M}/E]$ is a strictly pure substitution, it follows from lemma 4.5 that $[\underline{X}\ \overline{M}/E](P')$ is pure.

Case: $\mathsf{L}\Pi\Sigma$: analog

Case: case$'$: We can assume Γ to be pure, therefore $[P/X]$ is strictly pure and by lemma 4.10 we obtain $[P/X](\Gamma)$ is pure. From lemma 4.7 we obtain that $[P/X](G)$ is pure. By assumption $\overline{A_G}$ is pure, therefore B is a pure type. $[\eta](A_G{}')$ is pure by the side condition of the rule, therefore $\eta|_{Free(A_G{}')}$ is strictly pure by lemma 4.4. Since $\eta = \eta|_{Free(A_G{}')}$, η is a strictly pure substitution. Consequently for all i: $[\eta](P^{(i)})$ is pure, and therefore the proof term is a pure program.

□

Theorem 4.13 (Generalized Purity Preservation) *Let Γ be a pure context, G a pure formula, P a program, and $\mathcal{D}$ a derivation of $\mathcal{D} :: \Gamma \vdash_\Sigma P \in G$ in the inference system of pure MLF without Cut. Then P is pure.*

Proof: By induction over the derivation $\mathcal{D}$:

Case: id preserves purity: Under the assumption that $\Gamma_1, X \in C, \Gamma_2$ is pure, it follows trivially that X is pure and C is pure.

Case: const preserves purity because Γ is assumed to be pure.

Case: R1: analog.

Case: R$\wedge$: Γ pure and G_1, G_2 pure implies that P_1, P_2 pure, which implies (`pair` P_1 P_2), $G_1 \wedge G_2$ pure.

Case: R$\vee_1$: analog

Case: R$\vee_2$: analog

Case: R $\rightarrow$: Γ pure and D, G pure, therefore $\Gamma, X \in D$ pure. Induction hypothesis, gives as a pure P, and therefore (`fun` $X.P$) is pure.

Case: R$\forall$: Γ pure and A_D, G pure, therefore $\Gamma, Y \in \overline{A_D}$ pure. Since $[Y/X](G)$ is pure, induction hypothesis yields a pure program $[Y/X](P)$. Therefore (`fun` $X.P$) is pure.

Case: R$\exists'$: Γ pure, A_G, G pure. Y as the result of the first premiss is pure. Since $[Y/X]$ is strict, [Y/X](G) is pure, which yields a pure P. Therefore (`inx` Y P) is pure.

Case: R$\exists''$: Γ pure, A_G, G pure. $\overline{M}$ as the result of the first premiss is pure. Since $[\overline{M}/X]$ is strict, $[\overline{M}/X](G)$ is pure, which yields a pure P. Therefore (`inx` $\overline{M}$ P) is pure.

Case: rec: analog to R $\rightarrow$

Case: L$\wedge$: $\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2$ is pure, therefore $\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, X_1 \in D_1, X_2 \in D_2$ is pure. G is pure by assumption, therefore P is pure, which yields (`case` X `of` (`pair` X_1 X_2) $\Rightarrow P$) to be pure.

Case: L$\vee$: analog to L$\wedge$.

Case: L $\rightarrow$: analog to L$\wedge$.

Case: L$\forall'$: $\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2$ is pure, A_G is pure. Y_1 as a result of the first premiss is pure. Since $[Y_1/Y]$ is strict, and D is pure, $[Y_1/Y](D)$ is pure. Induction hypothesis on the second premiss yields P_2 is pure, and therefore (`let` (`app` X Y_1) `be` Z `in` P_2) is pure.

Case: L$\forall''$: $\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2$ is pure, A_G is pure. $\overline{M}$ as a result of the first premiss is pure. Since $[\overline{M}/Y]$ is strict, and D is pure, $[\overline{M}/Y](D)$ is pure. Induction hypothesis on the second premiss yields P_2 is pure, and therefore (`let` (`app` X $\overline{M}$) `be` Y `in` P_2) is pure.

Case: L$\exists$: analog to L$\wedge$.

Case: LΠ: $\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2$ is pure by assumption, therefore $\overline{M}$ is pure. $\overline{\Pi x : A_G.A_D}$ is pure, M is pure, therefore $\{M/x\}_{\text{type}}(A_D)$ is pure. therefore the context for the second premiss is pure, and the induction hypothesis yields P pure. Since $[\underline{X}\,\overline{M}/Y]$ is strict, $[\underline{X}\,\overline{M}/Y](P)$ is pure.

Case: L$\Pi\Sigma$: analog

Case: case$'$: $[P/X](\Gamma)$ is pure. If $X \in Free(\Gamma)$ then Γ is pure by lemma 4.6, else Γ is pure. If $X \in Free(G)$ then G is pure by lemma 4.8, else G is pure. A_G is pure and $A_G{}'$ is pure, and therefore $[\eta](A_G{}')$ is pure. By lemma 4.4 we obtain, that $\eta = \eta|_{Free(A_G{}')}$ is strict. We have $[\eta](\Gamma')$ is pure (because Γ is pure) and because of lemma 4.11 we obtain Γ' is pure. The same argument holds for $[\eta](G')$, which is pure since G is pure. Because of 4.8 we know that G' is pure, too. Because of construction $\Theta^{(i)}$ is pure for all i. Thus, $\Theta^{(i)}(\Gamma')$ is pure. $\Lambda^{(i)}$ is pure, too, because of construction. And finally $\Theta^{(i)}(G')$ is pure because G is pure. Therefore we can apply the induction hypothesis and obtain that the $P^{(i)}$'s are pure for all i. Therefore the $[\eta](P^{(i)})$ are pure, and hence the proof term

$$\left(\begin{array}{ll} \multicolumn{2}{l}{\texttt{case } P \texttt{ of}} \\ & \overline{c_1\, \underline{X_1^{(1)}} \ldots \underline{X_{m_1}^{(1)}}} \Rightarrow [\eta](P^{(1)}) \\ & \ldots \\ \mid & \overline{c_n\, \underline{X_1^{(n)}} \ldots \underline{X_{m_n}^{(n)}}} \Rightarrow [\eta](P^{(n)}) \end{array}\right)$$

is a pure program.

$\square$

Appendix C

Local reductions

Lemma 4.15 (Substitution Effects) *Let D be a data formula, P a program, K a kind, M an object and A_P an atomic type, A_G a goal type and A_D a data type. Let σ be a strict substitution, $Free(\sigma) \cap sup(\Gamma) = \emptyset$ and Λ be a context with $\vdash \Lambda$ ctx which introduces the new variables used in σ. Let $\Gamma' = \Lambda, [\sigma](\Gamma)$ and $\Delta' = [\sigma](\Delta)$. Then for all Γ meta context and Δ object context:*

$$\begin{array}{rcl}
\Gamma; \Delta \vdash_\Sigma K \; kind & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](K) \; kind \\
\Gamma; \Delta \vdash_\Sigma A_P : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) \; [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma A_G : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_G) \; [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma A_D : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_D) \; [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma M : A_P & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) \; [\sigma](A_P) \\
\Gamma; \Delta \vdash_\Sigma M : A_G & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) \; [\sigma](A_G) \\
\Gamma; \Delta \vdash_\Sigma M : A_D & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) \; [\sigma](A_D) \\
\Gamma \vdash_\Sigma D \; data & \Rightarrow & \Gamma' \vdash_\Sigma [\sigma](D) \; data \\
\Gamma \vdash_\Sigma P \in G & \Rightarrow & \Gamma' \vdash_\Sigma [\sigma](P) \in [\sigma](G) \\
\vdash_\Sigma \Gamma \; ctx & \Rightarrow & \vdash_\Sigma \Gamma' \; ctx
\end{array}$$

Proof: by mutual induction on the participating derivations: Note, $\mathcal{D}'_1$, $\mathcal{D}'_2$ always refer to the derivations we obtain by applying induction hypothesis to the derivation of the premiss of the rules.

Cases for $\Gamma; \Delta \vdash_\Sigma K$ kind:

Case: kindpi

$$\dfrac{\overset{\mathcal{D}'_1}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](A_G) : \text{type}} \qquad \overset{\mathcal{D}'_2}{\Gamma'; \Delta', x : [\sigma](A_G) \vdash_\Sigma [\sigma](K) \text{ kind}}}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](\Pi x : A_G.\, K) \text{ kind}} \text{kindpi}$$

other cases: analog or trivial

Cases for $\Gamma; \Delta \vdash_\Sigma A_P : K$:

Case: typeatomapp:

$$\dfrac{\overset{\mathcal{D}_1'}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) : \Pi x : [\sigma](A_G).\, [\sigma](K)} \qquad \overset{\mathcal{D}_2'}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_G)}}{\Gamma'; \Delta' \vdash_\Sigma ([\sigma](A_G)\ [\sigma](M)) : [\sigma](\{M/x\}(K))}\,\mathsf{typeatomapp}$$

Case: typeatomequiv:

$$\dfrac{\overset{\mathcal{D}_1'}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) : [\sigma](K)} \qquad [\sigma](K) \equiv [\sigma](K') \qquad \overset{\mathcal{D}_2'}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](K') : \mathrm{kind}}}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) : [\sigma](K')}\,\mathsf{typeatomequiv}$$

Cases for $\Gamma; \Delta \vdash_\Sigma A_G : K$:

All cases: same as $\Gamma; \Delta \vdash_\Sigma A_P : K$

Cases for $\Gamma; \Delta \vdash_\Sigma A_D : K$:

Case: typedatapi:

$$\dfrac{\overset{\mathcal{D}_1'}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](A_G) : \mathrm{type}} \qquad \overset{\mathcal{D}_2'}{\Gamma'; \Delta', x : [\sigma](A_G) \vdash_\Sigma [\sigma](A_D) : \mathrm{type}}}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](\Pi x : A_G.\, A_D) : \mathrm{type}}\,\mathsf{typedatapi}$$

other cases: analog or trivial or as $\Gamma; \Delta \vdash_\Sigma A_P : K$

Cases for $\Gamma; \Delta \vdash_\Sigma M : A_P$,$\Gamma; \Delta \vdash_\Sigma M : A_G$ and $\Gamma; \Delta \vdash_\Sigma M : A_D$:

Case: objdatapi

$$\dfrac{\overset{\mathcal{D}_1'}{\Gamma'; \Delta', x : [\sigma](A_G) \vdash_\Sigma [\sigma](M) : [\sigma](A_D)}}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](\lambda x : A_G.\, M) : [\sigma](\Pi x : A_G.\, A_D)}\,\mathsf{objdatapi}$$

Case: objdataapp

$$\dfrac{\overset{\mathcal{D}_1'}{\Gamma'; \Delta' \vdash_\Sigma M_1 : \Pi x : [\sigma](A_G).\, [\sigma](A_D)} \qquad \overset{\mathcal{D}_2'}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](M_2) : [\sigma](A_G)}}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](M_1\ M_2) : [\sigma](A_D)}\,\mathsf{objdataapp}$$

Case: objdataequiv

$$\dfrac{\overset{\mathcal{D}_1'}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_D)} \qquad [\sigma](A_D) \equiv [\sigma](A_D') \qquad \overset{\mathcal{D}_2'}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](A_D') : \mathrm{type}}}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_D')}\,\mathsf{objdataequiv}$$

Case: objgoalprgl

$$\dfrac{\overset{\mathcal{D}'_1}{\Gamma' \vdash_\Sigma [\sigma](P) \in \overline{[\sigma](A_G)}}}{\Gamma'; \Delta' \vdash_\Sigma [\sigma](\underline{P}) : [\sigma](A_G)}\ \textsf{objgoalprgl}$$

other cases: analog or trivial

Cases for $\Gamma \vdash_\Sigma D$ data:

Case: dataforall

$$\dfrac{\overset{\mathcal{D}'_1}{\Gamma', \cdot \vdash_\Sigma [\sigma](A_G) : \text{type}} \qquad \overset{\mathcal{D}'_2}{\Gamma', X \in \overline{[\sigma](A_G)} \vdash_\Sigma [\sigma](D) \text{ data}}}{\Gamma \vdash_\Sigma [\sigma](\forall X : A_G.D) \text{ data}}\ \textsf{dataforall}$$

Case: dataand

$$\dfrac{\overset{\mathcal{D}'_1}{\Gamma \vdash_\Sigma [\sigma](D_1) \text{ data}} \qquad \overset{\mathcal{D}'_2}{\Gamma \vdash_\Sigma [\sigma](D_2) \text{ data}}}{\Gamma \vdash_\Sigma [\sigma](D_1 \wedge D_2) \text{ data}}\ \textsf{dataand}$$

Case: datatype

$$\dfrac{\overset{\mathcal{D}'_1}{\Gamma'; \cdot \vdash_\Sigma [\sigma](K) \text{ kind}} \qquad \overset{\mathcal{D}'_2}{\Gamma'; \cdot \vdash_\Sigma [\sigma](A_D) : [\sigma](K)}}{\Gamma' \vdash_\Sigma [\sigma](\overline{A_D}) \text{ data}}\ \textsf{datatype}$$

other cases: analog or trivial

Cases for $\vdash_\Sigma \Gamma$ ctx:

Case: ctxcons

$$\dfrac{\overset{\mathcal{D}'_1}{\vdash_\Sigma \Gamma' \text{ ctx}} \qquad \overset{\mathcal{D}'_2}{\Gamma' \vdash_\Sigma [\sigma](D) \text{ data}}}{\vdash_\Sigma \Gamma', X \in [\sigma](D) \text{ ctx}}\ \textsf{ctxcons}$$

other cases: analog or trivial

Cases for $\Gamma \vdash_\Sigma P \in G$: straightforward

□

Lemma 4.14 (Context extension) *Let Γ_1, Γ_2 be contexts, s.t. $\vdash_\Sigma \Gamma_1, \Gamma_2$ ctx. Let D be formula, s.t. $\Gamma_1 \vdash_\Sigma D$ data, then $\vdash_\Sigma \Gamma_1, X \in D, \Gamma_2$ ctx*

Proof: by structural induction over the form of Γ_2:

Case: $\Gamma_2 = \cdot$: We have $\vdash_\Sigma \Gamma_1$ ctx. Since $\mathcal{E} :: \Gamma_1 \vdash_\Sigma D$ data we have $\vdash_\Sigma \Gamma_1, X \in D$ ctx

Case: $\Gamma_2 = \Gamma_2', X \in D$: By inversion we obtain $\vdash_\Sigma \Gamma_1, \Gamma_2'$ ctx. By induction hypothesis, we obtain $\vdash_\Sigma \Gamma_1, X \in D, \Gamma_2'$ ctx and by application of the context formation rule, we obtain $\vdash_\Sigma \Gamma_1, X \in D, \Gamma_2$ ctx.

□

Lemma 4.16 (Weakening) *Let $\mathcal{D} :: \Gamma_1, \Gamma_2 \vdash_\Sigma P \in G$, $\mathcal{E} :: \Gamma_1 \vdash_\Sigma D'$ data and $X' \notin dom(\Gamma_1, \Gamma_2)$, then $\mathcal{D}[\Gamma_1 \bigvee X \in D'] :: \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma P \in G$ where X' is new meta variable and D' is a data formula, depending only on variables in Γ_1.*

Proof: by induction over the derivation $\mathcal{D}$:

Case: id

$\vdash_\Sigma \Gamma_1, \Gamma_1', X \in C, \Gamma_2$ ctx	by assumption
$\Rightarrow \quad \vdash_\Sigma \Gamma_1, X' \in D', \Gamma_1', X \in C, \Gamma_2$ ctx	bylemma 4.14 and by assumption
$\Rightarrow \quad \Gamma_1, X' \in D', \Gamma_1', X \in C, \Gamma_2 \vdash_\Sigma X \in C$	Apply id

alternative analog

Case: R1: follows directly from lemma 4.14 and assumption

Case: const: follows directly from lemma 4.14 and assumption

Case: R∧

$\Gamma_1, \Gamma_2 \vdash_\Sigma P_1 \in G_1$	by assumption
$\Rightarrow \quad \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma P_1 \in G_1$	by hyp.
$\Gamma_1, \Gamma_2 \vdash_\Sigma P_2 \in G_2$	by assumption
$\Rightarrow \quad \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma P_2 \in G_2$	by hyp.
$\Rightarrow \quad \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma (\texttt{pair}\ P_1\ P_2) \in G_1 \wedge G_2$	Apply R∧

Case: R$\vee_1$

$\Gamma_1, \Gamma_2 \vdash_\Sigma P \in G_1$	by assumption
$\Rightarrow \quad \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma P \in G_1$	by hyp.
$\Rightarrow \quad \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma (\texttt{inl}\ P) \in G_1 \vee G_2$	Apply R$\vee_1$

Case: R$\vee_2$

$\Gamma_1, \Gamma_2 \vdash_\Sigma P \in G_2$	by assumption
$\Rightarrow \quad \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma P \in G_2$	by hyp.
$\Rightarrow \quad \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma (\texttt{inl}\ P) \in G_1 \vee G_2$	Apply R$\vee_2$

Case: R $\rightarrow$

$\Gamma_1, \Gamma_2, X \in D \vdash_\Sigma P \in G$ by assumption
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2, X \in D \vdash_\Sigma P \in G$ by hyp.
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma (\mathtt{fun}\, X.P) \in D \rightarrow G$ Apply R $\rightarrow$

Case: R$\forall$

$\Gamma_1, \Gamma_2, Y \in \overline{A_D} \vdash_\Sigma [Y/X]P \in [Y/X](G)$ by assumption
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2, Y \in \overline{A_D} \vdash_\Sigma [Y/X]P \in [Y/X](G)$ by hyp.
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma (\mathtt{fun}\, Y.P) \in \forall X : A_D.G$ Apply R$\forall$

Case: R$\exists$

$\Gamma_1, \Gamma_2 \vdash_\Sigma P' \in \overline{A_G}$ by assumption
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma P' \in \overline{A_G}$ by hyp.
$\Gamma_1, \Gamma_2 \vdash_\Sigma P \in [P'/X](G)$ by assumption
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma P \in [P'/X](G)$ by hyp.
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma (\mathtt{inx}\; P'\; P) \in \exists X : A_G.G$ Apply R$\exists$

Case: rec

$\Gamma_1, \Gamma_2, X \in C \vdash_\Sigma P \in C$ by assumption
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2, X \in C \vdash_\Sigma P \in C$ by hyp.
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma (\mathtt{rec}\, X.P) \in C$ Apply rec

Case: L$\wedge$

$\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, \Gamma_3, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G$ by assumption
$\Rightarrow \;\; \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, X' \in D', \Gamma_3, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G$ by hyp.
$\Rightarrow \;\; \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma (\mathtt{case}\; X\; \mathtt{of}\; (\mathtt{pair}\; X_1\; X_2) \Rightarrow P) \in G$ Apply L$\wedge$

$\Gamma_1, \Gamma_2, X \in D_1 \wedge D_2, \Gamma_3, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G$ by assumption
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2, X \in D_1 \wedge D_2, \Gamma_3, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G$ by hyp.
$\Rightarrow \;\; \Gamma_1, X' \in D', \Gamma_2, X \in D_1 \wedge D_2, \Gamma_3 \vdash_\Sigma (\mathtt{case}\; X\; \mathtt{of}\; (\mathtt{pair}\; X_1\; X_2) \Rightarrow P) \in G$ Apply L$\wedge$

Case: L$\vee$

$\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, \Gamma_3, X_1 \in D_1 \vdash_\Sigma P_1 \in G$ by assumption
$\Rightarrow\ \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, X' \in D', \Gamma_3, X_1 \in D_1 \vdash_\Sigma P_1 \in G$ by hyp.
$\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, \Gamma_3, X_2 \in D_2 \vdash_\Sigma P_2 \in G$ by assumption
$\Rightarrow\ \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, X' \in D', \Gamma_3, X_2 \in D_2 \vdash_\Sigma P_2 \in G$ by hyp.
$\Rightarrow\ \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma \left(\begin{array}{ll} \mathtt{case}\ X\ \mathtt{of} & (\mathtt{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\mathtt{inr}\ X_2) \Rightarrow P_2 \end{array}\right) \in G$ Apply L$\vee$

$\Gamma_1, \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, X_1 \in D_1 \vdash_\Sigma P_1 \in G$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, X_1 \in D_1 \vdash_\Sigma P_1 \in G$ by hyp.
$\Gamma_1, \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, X_2 \in D_2 \vdash_\Sigma P_2 \in G$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, X_2 \in D_2 \vdash_\Sigma P_2 \in G$ by hyp.
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in D_1 \vee D_2, \Gamma_3 \vdash_\Sigma \left(\begin{array}{ll} \mathtt{case}\ X\ \mathtt{of} & (\mathtt{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\mathtt{inr}\ X_2) \Rightarrow P_2 \end{array}\right) \in G$ Apply L$\vee$

Case: L$\rightarrow$

$\Gamma_1, X \in G_1 \rightarrow D, \Gamma_2, \Gamma_3 \vdash_\Sigma P \in G_1$ by assumption
$\Rightarrow\ \Gamma_1, X \in G_1 \rightarrow D, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma P \in G_1$ by hyp.
$\Gamma_1, X \in G_1 \rightarrow D, \Gamma_2, \Gamma_3, Y \in D \vdash_\Sigma P' \in G_2$ by assumption
$\Rightarrow\ \Gamma_1, X \in G_1 \rightarrow D, \Gamma_2, X' \in D', \Gamma_3, Y \in D \vdash_\Sigma P' \in G_2$ by hyp.
$\Rightarrow\ \Gamma_1, X \in G_1 \rightarrow D, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma (\mathtt{let}\ (\mathtt{app}\ X\ P)\ \mathtt{be}\ Y\ \mathtt{in}\ P') \in G_2$ Apply L$\rightarrow$

$\Gamma_1, \Gamma_2, X \in G_1 \rightarrow D, \Gamma_3 \vdash_\Sigma P \in G_1$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in G_1 \rightarrow D, \Gamma_3 \vdash_\Sigma P \in G_1$ by hyp.
$\Gamma_1, \Gamma_2, X \in G_1 \rightarrow D, \Gamma_3, Y \in D \vdash_\Sigma P' \in G_2$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in G_1 \rightarrow D, \Gamma_3, Y \in D \vdash_\Sigma P' \in G_2$ by hyp.
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in G_1 \rightarrow D, \Gamma_3 \vdash_\Sigma (\mathtt{let}\ (\mathtt{app}\ X\ P)\ \mathtt{be}\ Y\ \mathtt{in}\ P') \in G_2$ Apply L$\rightarrow$

Case: L$\forall$

$\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \Gamma_3 \vdash_\Sigma P_1 \in \overline{A_G}$ by assumption
$\Rightarrow\ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma P_1 \in \overline{A_G}$ by hyp.
$\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \Gamma_3, Z \in [P_1/Y](D) \vdash_\Sigma P_2 \in G$ by assumption
$\Rightarrow\ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, X' \in D', \Gamma_3, Z \in [P_1/Y](D) \vdash_\Sigma P_2 \in G$ by hyp.
$\Rightarrow\ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma (\mathtt{let}\ (\mathtt{app}\ X\ P_1)\ \mathtt{be}\ Z\ \mathtt{in}\ P_2) \in G$ Apply L$\forall$

$\Gamma_1, \Gamma_2, X \in \forall Y : A_G.D, \Gamma_3 \vdash_\Sigma P_1 \in \overline{A_G}$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in \forall Y : A_G.D, \Gamma_3 \vdash_\Sigma P_1 \in \overline{A_G}$ by hyp.

$\Gamma_1, \Gamma_2, X \in \forall Y : A_G.D, \Gamma_3, Z \in [P_1/Y](D) \vdash_\Sigma P_2 \in G$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in \forall Y : A_G.D, \Gamma_3, Z \in [P_1/Y](D) \vdash_\Sigma P_2 \in G$ by hyp.
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in \forall Y : A_G.D, \Gamma_3 \vdash_\Sigma (\mathtt{let}\ (\mathtt{app}\ X\ P_1)\ \mathtt{be}\ Z\ \mathtt{in}\ P_2) \in G$ Apply L$\forall$

Case: L$\exists$

$\Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, \Gamma_3, X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D) \vdash_\Sigma P \in G$ by assumption
$\Rightarrow\ \Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, X' \in D', \Gamma_3, X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D) \vdash_\Sigma P \in G$ by hyp.
$\Rightarrow\ \Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma (\mathtt{case}\ X\ \mathtt{of}\ (\mathtt{inx}\ X_1\ X_2) \Rightarrow P) \in G$ Apply L$\exists$

$\Gamma_1, \Gamma_2, X \in \exists Y : A_D.D, \Gamma_3, X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D) \vdash_\Sigma P \in G$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in \exists Y : A_D.D, \Gamma_3, X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D) \vdash_\Sigma P \in G$ by hyp.
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in \exists Y : A_D.D, \Gamma_3 \vdash_\Sigma (\mathtt{case}\ X\ \mathtt{of}\ (\mathtt{inx}\ X_1\ X_2) \Rightarrow P) \in G$ Apply L$\exists$

Case: LΠ

$\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, \Gamma_3 \vdash_\Sigma \overline{M} \in \overline{A_G}$ by assumption
$\Rightarrow\ \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma \overline{M} \in \overline{A_G}$ by hyp.
$\Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, \Gamma_3, Y \in \overline{\{M/x\}_{\text{type}}(A_D)} \vdash_\Sigma P \in G$ by assumption
$\Rightarrow\ \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, X' \in D', \Gamma_3, Y \in \overline{\{M/x\}_{\text{type}}(A_D)} \vdash_\Sigma P \in G$ by hyp.
$\Rightarrow\ \Gamma_1, X \in \overline{\Pi x : A_G.A_D}, \Gamma_2, X' \in D', \Gamma_3 \vdash_\Sigma [\overline{(\underline{X}\ M)}/Y](P) \in G$ Apply LΠ

$\Gamma_1, \Gamma_2, X \in \overline{\Pi x : A_G.A_D}, \Gamma_3 \vdash_\Sigma \overline{M} \in \overline{A_G}$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in \overline{\Pi x : A_G.A_D}, \Gamma_3 \vdash_\Sigma \overline{M} \in \overline{A_G}$ by hyp.
$\Gamma_1, \Gamma_2, X \in \overline{\Pi x : A_G.A_D}, \Gamma_3, Y \in \overline{\{M/x\}_{\text{type}}(A_D)} \vdash_\Sigma P \in G$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in \overline{\Pi x : A_G.A_D}, \Gamma_3, Y \in \overline{\{M/x\}_{\text{type}}(A_D)} \vdash_\Sigma P \in G$ by hyp.
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, X \in \overline{\Pi x : A_G.A_D}, \Gamma_3 \vdash_\Sigma [\overline{(\underline{X}\ M)}/Y](P) \in G$ Apply LΠ

Case: LΠV analog

Case: L$\Pi\Sigma$

$\Gamma_1, \Gamma_2 \vdash_\Sigma \overline{M} \in \overline{A_G}$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma \overline{M} \in \overline{A_G}$ by hyp.
$\Gamma_1, \Gamma_2, Y \in \overline{\{M/x\}_{\text{type}}(A_D)} \vdash_\Sigma P' \in G$ by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2, Y \in \overline{\{M/x\}_{\text{type}}(A_D)} \vdash_\Sigma P' \in G$ by hyp.
$\Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma [\overline{(c\ M)}/Y](P') \in G$ Apply L$\Pi\Sigma$

Case: LΠΣV

Case: case

$\Gamma_1, \Gamma_2 \vdash P \in \overline{A_G}$	by assumption
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2 \vdash P \in \overline{A_G}$	by hyp.
$\Gamma_1 \vdash D'$ formula	by assumption
$\Lambda^{(i)}, \Theta^{(i)}(\Gamma_1) \vdash \Theta^{(i)}(D')$ formula	by lemma 3.32 (3.8, 3.9), and lemma 4.15
For all $i \leq n$	
$\Lambda^{(i)}, \Theta^{(i)}(\Gamma_1, \Gamma_2) \vdash_\Sigma P^{(i)} \in \Theta^{(i)}(G)$	by assumption
$\Rightarrow\ \Lambda^{(i)}, \Theta^{(i)}(\Gamma_1), X' \in \Theta^{(i)}(D'), \Theta^{(i)}(\Gamma_2) \vdash_\Sigma P^{(i)} \in \Theta^{(i)}(G)$	by hyp.
$\Rightarrow\ \Lambda^{(i)}, \Theta^{(i)}(\Gamma_1, X' \in D', \Gamma_2) \vdash_\Sigma P^{(i)} \in \Theta^{(i)}(G)$	by def. subst.
$\Rightarrow\ \Gamma_1, X' \in D', \Gamma_2 \vdash_\Sigma \left(\begin{array}{l} \mathsf{case}\ P\ \mathsf{of} \\ \quad \overline{c_1\ \underline{Y_1^{(1)}}..\underline{Y_{m_1}^{(1)}} \Rightarrow [\eta](P^{(1)})} \\ \quad \ldots \\ \mid\ \overline{c_n\ \underline{Y_1^{(n)}}..\underline{Y_{m_n}^{(n)}} \Rightarrow [\eta](P^{(n)})} \end{array}\right) \in G$	Apply case

□

Lemma 4.17 (Contraction:) *Let D, D' be data formulae, K a kind, A_P an atomic type, A_G a goal type, A_D a data type, M an object and P a program. Then the following holds: For all meta contexts $\Gamma_1, \Gamma_2, \Gamma_3$, and for all object context Δ: Let $\Gamma = \Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3$, and $\Gamma' = \Gamma_1, U \in D', \Gamma_2, [U/V](\Gamma_3)$ and let $\Delta' = [U/V](\Delta)$ and $\sigma = [U/V]$. Then we have:*

$$\begin{array}{rcl}
\Gamma; \Delta \vdash_\Sigma K\ \mathit{kind} & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](K)\ \mathit{kind} \\
\Gamma; \Delta \vdash_\Sigma A_P : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) : [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma A_G : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_D) : [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma A_D : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_G) : [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma M : A_P & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_P) \\
\Gamma; \Delta \vdash_\Sigma M : A_D & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_D) \\
\Gamma; \Delta \vdash_\Sigma M : A_G & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_G) \\
\Gamma \vdash_\Sigma D\ \mathit{data} & \Rightarrow & \Gamma' \vdash_\Sigma [\sigma](D)\ \mathit{data} \\
\Gamma \vdash_\Sigma P \in G & \Rightarrow & \Gamma' \vdash_\Sigma [\sigma](P) \in [\sigma](G) \\
\vdash_\Sigma \Gamma\ \mathit{ctx} & \Rightarrow & \vdash_\Sigma \Gamma'\ \mathit{ctx}
\end{array}$$

Proof: by mutual induction on the participating derivations: Note, $\mathcal{D}_1'$, $\mathcal{D}_2'$ always refer to the derivations we obtain by applying induction hypothesis to the derivation of the premiss of the rules.

Cases for $\Gamma; \Delta \vdash_\Sigma K$ kind: same as in proof for lemma 4.15.

Cases for $\Gamma; \Delta \vdash_\Sigma A_P : K$: same as in proof for lemma 4.15.

Cases for $\Gamma;\Delta \vdash_\Sigma A_G : K$**:** same as in proof for lemma 4.15.

Cases for $\Gamma;\Delta \vdash_\Sigma A_D : K$**:** same as in proof for lemma 4.15.

Cases for $\Gamma;\Delta \vdash_\Sigma M : A_P$**:** same as in proof for lemma 4.15.

Cases for $\Gamma;\Delta \vdash_\Sigma M : A_G$**:** same as in proof for lemma 4.15.

Cases for $\Gamma;\Delta \vdash_\Sigma M : A_D$**:** same as in proof for lemma 4.15.

Cases for $\Gamma \vdash_\Sigma D$ **data:** same as in proof for lemma 4.15.

Cases for $\vdash_\Sigma \Gamma$ **ctx:** same as in proof for lemma 4.15.

Cases for $\Gamma \vdash_\Sigma P \in G$**:**

Case: id

$\vdash_\Sigma \Gamma_1, X \in C, \Gamma_2, U \in D', \Gamma_3, V \in D', \Gamma_4$ ctx	Assumption
$\Rightarrow\ \vdash_\Sigma \Gamma_1, X \in C, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4)$ ctx	by hyp.
$\Rightarrow\ \Gamma_1, X \in C, \Gamma_2, U \in D', \Gamma_3, \sigma(\Gamma_4) \vdash_\Sigma X \in [\sigma](C)$	Apply id
$\equiv\ \Gamma_1, X \in C, \Gamma_2, U \in D', \Gamma_3, \sigma(\Gamma_4) \vdash_\Sigma X \in C$	trivial

$\vdash_\Sigma \Gamma_1, U \in D', \Gamma_2, X \in C, \Gamma_3, V \in D', \Gamma_4$ ctx	Assumption
$\Rightarrow\ \vdash_\Sigma \Gamma_1, U \in D', \Gamma_2, D \in C, \Gamma_3, [\sigma](\Gamma_4)$ ctx	by hyp.
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, X \in C, \Gamma_3, \sigma(\Gamma_4) \vdash_\Sigma X \in [\sigma](C)$	Apply id
$\equiv\ \Gamma_1, U \in D', \Gamma_2, X \in C, \Gamma_3, \sigma(\Gamma_4) \vdash_\Sigma X \in C$	trivial

$\vdash_\Sigma \Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3, X \in C, \Gamma_4$ ctx	Assumption
$\vdash_\Sigma \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](C), [\sigma](\Gamma_4)$ ctx	Ih
$\equiv\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](C), [\sigma](\Gamma_4) \vdash_\Sigma X \in [\sigma](C)$	Apply id

Case: R1, const: trivial

Case: R$\wedge$

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3 \vdash_\Sigma P_1 \in G_1$	by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](P_1) \in [\sigma](G_1)$	by hyp.
$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3 \vdash_\Sigma P_2 \in G_2$	by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](P_2) \in [\sigma](G_2)$	by hyp.
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](\texttt{pair}\ P_1\ P_2) \in [\sigma](G_1 \wedge G_2)$	Apply R$\wedge$

Case: R$\vee_1$

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3 \vdash_\Sigma P \in G_1$	by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](P) \in [\sigma](G_1)$	by hyp.
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](\texttt{inl}\ P) \in [\sigma](G_1 \vee G_2)$	Apply R$\vee_1$

Case: $\mathsf{R}\vee_2$

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3 \vdash_\Sigma P \in G_2$ by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma , [\sigma](P) \in, [\sigma](G_2)$ by hyp.
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma , [\sigma](\texttt{inl}\ P) \in, [\sigma](G_1 \vee G_2)$ Apply $\mathsf{R}\vee_2$

Case: $\mathsf{R}\rightarrow$

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3, X \in D \vdash_\Sigma P \in G$ by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in D \vdash_\Sigma [\sigma](P) \in [\sigma](G)$ by hyp.
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](\texttt{fun}\, X.P) \in [\sigma](D \rightarrow G)$ Apply $\mathsf{R}\rightarrow$

Case: $\mathsf{R}\forall$

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3, Y \in \overline{A_D} \vdash_\Sigma [Y/X]P \in [Y/X](G)$ by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), Y \in \overline{[\sigma](A_D)} \vdash_\Sigma [\sigma]([Y/X](P)) \in [\sigma]([Y/X](G))$ by hyp.
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), Y \in \overline{[\sigma](A_D)} \vdash_\Sigma [Y/X]([\sigma](P)) \in [Y/X]([\sigma](G))$ trivial
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](\texttt{fun}\, X.P) \in [\sigma](\forall X : A_D.G)$ Apply $\mathsf{R}\forall$, by def. subst.

Case: $\mathsf{R}\exists$

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3 \vdash_\Sigma P' \in \overline{A_G}$ by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](P') \in \overline{[\sigma](A_G)}$ by hyp.
$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3 \vdash_\Sigma P \in [P'/X](G)$ by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](P) \in [\sigma]([P'/X](G))$ by hyp.
$\equiv\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](P) \in [[\sigma](P')/X]([\sigma](G))$ trivial
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](\texttt{inx}\, P'\ P) \in [\sigma](\exists X : A_G.G)$ Apply $\mathsf{R}\exists$

Case: rec

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3, X \in C \vdash_\Sigma P \in G$ by assumption
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](C) \vdash_\Sigma [\sigma](P) \in [\sigma](C)$ by hyp.
$\Rightarrow\ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](\texttt{rec}\, X.P) \in [\sigma](C)$ Apply rec, by def. subst.

Case: $\mathsf{L}\wedge$

$\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, U \in D', \Gamma_3, V \in D', \Gamma_4, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G$ by assumption
$\Rightarrow\ \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4), X_1 \in [\sigma](D_1), X_2 \in [\sigma](D_2)$
$\vdash_\Sigma [\sigma](P) \in [\sigma](G)$ by hyp.
$\Rightarrow\ \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4), X_1 \in D_1, X_2 \in D_2$
$\vdash_\Sigma [\sigma](P) \in [\sigma](G)$ trivial
$\Rightarrow\ \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4)$
$\vdash_\Sigma [\sigma](\texttt{case}\, X\ \texttt{of}\ (\texttt{pair}\, X_1\ X_2) \Rightarrow P) \in [\sigma](G)$ Apply $\mathsf{L}\wedge$

$\Gamma_1, U \in D', \Gamma_2, X \in D_1 \wedge D_2, \Gamma_3, V \in D', \Gamma_4, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G$ by assumption
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, X \in D_1 \wedge D_2, \Gamma_3, [\sigma](\Gamma_4), X_1 \in D_1, X_2 \in D_2$
$\vdash_\Sigma [\sigma](P) \in [\sigma](G)$ by hyp.
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, X \in D_1 \wedge D_2, \Gamma_3, [\sigma](\Gamma_4)$
$\vdash_\Sigma [\sigma](\texttt{case}\, X \,\texttt{of}\, (\texttt{pair}\, X_1\, X_2) \Rightarrow P) \in [\sigma](G)$ Apply L$\wedge$

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3, X \in D_1 \wedge D_2, \Gamma_4, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G$ by assumption
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3, X \in D_1 \wedge D_2, \Gamma_4), X_1 \in [\sigma](D_1), X_2 \in [\sigma](D_2)$
$\vdash_\Sigma [\sigma](P) \in [\sigma](G)$ by hyp.
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](D_1) \wedge [\sigma](D_2), [\sigma](\Gamma_4)$
$\vdash_\Sigma [\sigma](\texttt{case}\, X \,\texttt{of}\, (\texttt{pair}\, X_1\, X_2) \Rightarrow P) \in [\sigma](G)$ Apply L$\wedge$
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](D_1 \wedge D_2), [\sigma](\Gamma_4)$
$\vdash_\Sigma [\sigma](\texttt{case}\, X \,\texttt{of}\, (\texttt{pair}\, X_1\, X_2) \Rightarrow P) \in [\sigma](G)$ trivial

Case: L$\vee$

$\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, U \in D', \Gamma_3, V \in D', \Gamma_4, X_1 \in D_1 \vdash_\Sigma P_1 \in G$ by assumption
$\Rightarrow$ $\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4), X_1 \in [\sigma](D_1) \vdash_\Sigma [\sigma](P_1) \in [\sigma](G)$ by hyp.
$\equiv$ $\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4), X_1 \in D_1 \vdash_\Sigma [\sigma](P_1) \in [\sigma](G)$ trivial
$\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, U \in D', \Gamma_3, V \in D', \Gamma_4, X_2 \in D_2 \vdash_\Sigma P_2 \in G$ by assumption
$\Rightarrow$ $\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4), X_2 \in [\sigma](D_2) \vdash_\Sigma [\sigma](P_2) \in [\sigma](G)$ by hyp.
$\equiv$ $\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4), X_2 \in D_2 \vdash_\Sigma [\sigma](P_2) \in [\sigma](G)$ trivial
$\Rightarrow$ $\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, U \in D', \Gamma_3, [\sigma](\Gamma_4)$
$$\vdash_\Sigma [\sigma]\begin{pmatrix} \texttt{case}\, X \,\texttt{of} & (\texttt{inl}\, X_1) \Rightarrow P_1 \\ | & (\texttt{inr}\, X_2) \Rightarrow P_2 \end{pmatrix} \in [\sigma](G) \qquad \text{Apply L}\vee$$

$\Gamma_1, U \in D', \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, V \in D', \Gamma_4, X_1 \in D_1 \vdash_\Sigma P_1 \in G$ by assumption
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, [\sigma](\Gamma_4), X_1 \in D_1 \vdash_\Sigma [\sigma](P_1) \in [\sigma](G)$ by hyp.
$\Gamma_1, U \in D', \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, V \in D', \Gamma_4, X_2 \in D_2 \vdash_\Sigma P_2 \in G$ by assumption
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, [\sigma](\Gamma_4), X_2 \in D_2 \vdash_\Sigma [\sigma](P_2) \in [\sigma](G)$ by hyp.
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, [\sigma](\Gamma_4)$
$$\vdash_\Sigma [\sigma]\begin{pmatrix} \texttt{case}\, X \,\texttt{of} & (\texttt{inl}\, X_1) \Rightarrow P_1 \\ | & (\texttt{inr}\, X_2) \Rightarrow P_2 \end{pmatrix} \in [\sigma](G) \qquad \text{Apply L}\vee$$

$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3, X \in D_1 \vee D_2, \Gamma_4, X_1 \in D_1 \vdash_\Sigma P_1 \in G$ by assumption
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3, X \in D_1 \vee D_2, \Gamma_4), X_1 \in [\sigma](D_1) \vdash_\Sigma [\sigma](P_1) \in [\sigma](G)$ by hyp.
$\equiv$ $\Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](D_1 \vee D_2), [\sigma](\Gamma_4), X_1 \in [\sigma](D_1)$
$\vdash_\Sigma [\sigma](P_1) \in [\sigma](G)$ trivial
$\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3, X \in D_1 \vee D_2, \Gamma_4, X_2 \in D_2 \vdash_\Sigma P_2 \in G$ by assumption
$\Rightarrow$ $\Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3, X \in D_1 \vee D_2, \Gamma_4), X_2 \in [\sigma](D_2) \vdash_\Sigma [\sigma](P_2) \in [\sigma](G)$ by hyp.
$\equiv$ $\Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](D_1 \vee D_2), [\sigma](\Gamma_4), X_2 \in [\sigma](D_2)$

$$
\begin{array}{lr}
\quad \vdash_\Sigma [\sigma](P_2) \in [\sigma](G) & \text{trivial} \\
\Rightarrow \ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](D_1) \vee [\sigma](D_2), [\sigma](\Gamma_4) & \\
\quad \vdash_\Sigma [\sigma]\left(\begin{array}{ll}\texttt{case}\ X\ \texttt{of} & (\texttt{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\texttt{inr}\ X_2) \Rightarrow P_2\end{array}\right) \in [\sigma](G) & \text{Apply L}\vee \\
\Rightarrow \ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3), X \in [\sigma](D_1 \vee D_2), [\sigma](\Gamma_4) & \\
\quad \vdash_\Sigma [\sigma]\left(\begin{array}{ll}\texttt{case}\ X\ \texttt{of} & (\texttt{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\texttt{inr}\ X_2) \Rightarrow P_2\end{array}\right) \in [\sigma](G) & \text{trivial}
\end{array}
$$

Cases: other left rules follow the same pattern.

Case: case: $\Gamma = \Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3$ Note that the declaration $X \in \overline{A_G}$ must be in Γ_1, Γ_2 or Γ_3.

$$
\begin{array}{lr}
[P/X](\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3)\ \text{ctx} & \text{by assumption} \\
\Rightarrow \ [P/X](\Gamma_1), U \in [P/X](D'), [P/X](\Gamma_2), V \in [P/X](D'), [P/X](\Gamma_3))\ \text{ctx} & \text{by def. subst.} \\
\Rightarrow \ [P/X](\Gamma_1'), U \in [P/X](D'), [P/X](\Gamma_2), [\sigma]([P/X](\Gamma_3)))\ \text{ctx} & \text{by hyp.} \\
\Gamma_1, U \in D', \Gamma_2, V \in D', \Gamma_3 \vdash_\Sigma P \in \overline{A_G} & \text{by assumption} \\
\Rightarrow \ \Gamma_1, U \in D', \Gamma_2, [\sigma](\Gamma_3) \vdash_\Sigma [\sigma](P) \in [\sigma](\overline{A_G}) & \text{by hyp.} \\
\equiv \ [\sigma](\Gamma_1, U \in D', \Gamma_2, \Gamma_3) \vdash_\Sigma [\sigma](P) \in [\sigma](\overline{A_G}) & \text{by def. subst.}
\end{array}
$$

Therefore $[\sigma](A_G) = [\sigma \circ \eta](A_G')$, $[\sigma](\Gamma) = [\sigma \circ \eta](\Gamma')$, $[\sigma](G) = [\sigma \circ \eta](G')$. Define $\eta' = \sigma \circ \eta$. Therefore all derivations for

$$\Lambda^{(i)}, \Theta^{(i)}(\Gamma') \vdash_\Sigma P^{(i)} \in \Theta^{(i)}(G')$$

are still premisses. The application of the rule yields:

$$
\begin{array}{l}
[[\sigma](P)/X]([\sigma](\Gamma_1, U \in D', \Gamma_2, \Gamma_3)) \\
\vdash_\Sigma \left(\begin{array}{l}\texttt{case}\ [\sigma](P)\ \texttt{of} \\ \quad \overline{c_1\ \underline{Y_1^{(1)}}...\underline{Y_{m_1}^{(1)}}} \Rightarrow [\sigma \circ \eta](P^{(1)}) \\ \quad ... \\ \mid \quad \overline{c_n\ \underline{Y_1^{(n)}}...\underline{Y_{m_n}^{(n)}}} \Rightarrow [\sigma \circ \eta](P^{(n)})\end{array}\right) \in [[\sigma](P)/X]([\sigma](G))
\end{array}
$$

which is equivalent to

$$
\begin{array}{l}
[\sigma]([P/X](\Gamma_1, U \in D'\Gamma_2, \Gamma_3)) \\
\vdash_\Sigma [\sigma] \left(\begin{array}{l}\texttt{case}\ P\ \texttt{of} \\ \quad \overline{c_1\ \underline{Y_1^{(1)}}...\underline{Y_{m_1}^{(1)}}} \Rightarrow [\eta](P^{(1)}) \\ \quad ... \\ \mid \quad \overline{c_n\ \underline{Y_1^{(n)}}...\underline{Y_{m_n}^{(n)}}} \Rightarrow [\eta](P^{(n)})\end{array}\right) \in [\sigma]([P/X](G))
\end{array}
$$

□

Lemma 4.19 (Substitution Lemma:) *Let D be data formulae, K a kind, A_P an atomic type, A_G goal type, A_D, A data types, M object and P a program. For $c : A \in \Sigma$ the following holds: For all meta contexts Γ_1, Γ_2, and for all object context Δ: Let $\Gamma = \Gamma_1, Z \in \overline{A}, \Gamma_2$, and $\Gamma' = \Gamma_1, [\overline{c}/Z](\Gamma_2)$ and let $\Delta' = [\overline{c}/Z](\Delta)$ and $\sigma = [\overline{c}/Z]$. Then we have:*

$$\begin{array}{rcl}
\Gamma; \Delta \vdash_\Sigma K\ kind & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](K)\ kind \\
\Gamma; \Delta \vdash_\Sigma A_P : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_P) : [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma A_G : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_G) : [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma A_D : K & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](A_D) : [\sigma](K) \\
\Gamma; \Delta \vdash_\Sigma M : A_P & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_P) \\
\Gamma; \Delta \vdash_\Sigma M : A_G & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_G) \\
\Gamma; \Delta \vdash_\Sigma M : A_D & \Rightarrow & \Gamma'; \Delta' \vdash_\Sigma [\sigma](M) : [\sigma](A_D) \\
\Gamma \vdash_\Sigma D\ data & \Rightarrow & \Gamma' \vdash_\Sigma [\sigma](D)\ data \\
\Gamma \vdash_\Sigma P \in G & \Rightarrow & \Gamma' \vdash_\Sigma [\sigma](P) \in [\sigma](G) \\
\vdash_\Sigma \Gamma\ ctx & \Rightarrow & \vdash_\Sigma \Gamma'\ ctx
\end{array}$$

Proof:

Cases for $\Gamma; \Delta \vdash_\Sigma K$ kind: same as in proof for lemma 4.15.

Cases for $\Gamma; \Delta \vdash_\Sigma A_P : K$: same as in proof for lemma 4.15.

Cases for $\Gamma; \Delta \vdash_\Sigma A_G : K$: same as in proof for lemma 4.15.

Cases for $\Gamma; \Delta \vdash_\Sigma A_D : K$: same as in proof for lemma 4.15.

Cases for $\Gamma; \Delta \vdash_\Sigma M : A_P$: same as in proof for lemma 4.15.

Cases for $\Gamma; \Delta \vdash_\Sigma M : A_G$: same as in proof for lemma 4.15.

Cases for $\Gamma; \Delta \vdash_\Sigma M : A_D$: same as in proof for lemma 4.15.

Cases for $\Gamma \vdash_\Sigma D$ data: same as in proof for lemma 4.15.

Cases for $\vdash_\Sigma \Gamma$ ctx: same as in proof for lemma 4.15.

Cases for $\Gamma \vdash_\Sigma P \in G$:

Case: All axioms and right rules are straightforward

Case: L$\wedge$

$$\begin{array}{llr}
& \Gamma_1, Z \in \overline{A}, \Gamma_2, X \in D_1 \wedge D_2, \Gamma_3, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G & \text{Ass.} \\
\Rightarrow & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1 \wedge D_2), \sigma(\Gamma_3), X_1 \in \sigma(D_1), X_2 \in \sigma(D_2) & \\
& \vdash_\Sigma \sigma(P) \in \sigma(G) & \text{I.H.} \\
\equiv & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1) \wedge \sigma(D_2), \sigma(\Gamma_3), X_1 \in \sigma(D_1), X_2 \in \sigma(G_2) & \\
& \vdash_\Sigma \sigma(P) \in \sigma(G) & \text{Def. subst}
\end{array}$$

$$\begin{array}{lll}
\Rightarrow & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1) \wedge \sigma(D_2), \sigma(\Gamma_3) & \\
 & \vdash_\Sigma (\texttt{case}\ X\ \texttt{of}\ (\texttt{pair}\ X_1\ X_2) \Rightarrow \sigma(P)) \in \sigma(G) & \text{Apply } \mathsf{L}\wedge \\
\equiv & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1 \wedge D_2), \sigma(\Gamma_3) & \\
 & \vdash_\Sigma \sigma(\texttt{case}\ X\ \texttt{of}\ (\texttt{pair}\ X_1\ X_2) \Rightarrow P) \in \sigma(G) & \text{Def. subst}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, Z \in \overline{A}, \Gamma_3, X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma P \in G} & \\
\Rightarrow & \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, \sigma(\Gamma_3), X_1 \in \sigma(D_1), X_2 \in \sigma(D_2) \vdash_\Sigma \sigma(P) \in \sigma(G) & \text{I.H.} \\
\Rightarrow & \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, \sigma(\Gamma_3), X_1 \in D_1, X_2 \in D_2 \vdash_\Sigma \sigma(P) \in \sigma(G) & \text{trivial} \\
\Rightarrow & \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, \sigma(\Gamma_3) & \\
 & \vdash_\Sigma (\texttt{case}\ X\ \texttt{of}\ (\texttt{pair}\ X_1\ X_2) \Rightarrow \sigma(P)) \in \sigma(G) & \text{Apply } \mathsf{L}\wedge \\
\equiv & \Gamma_1, X \in D_1 \wedge D_2, \Gamma_2, \sigma(\Gamma_3) & \\
 & \vdash_\Sigma \sigma(\texttt{case}\ X\ \texttt{of}\ (\texttt{pair}\ X_1\ X_2) \Rightarrow P) \in \sigma(G) & \text{Def. subst}
\end{array}$$

Case: $\mathsf{L}\vee$

$$\begin{array}{lll}
\multicolumn{2}{l}{\Gamma_1, Z \in \overline{A}, \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, X_1 \in D_1 \vdash_\Sigma P_1 \in G} & \text{Ass.} \\
\Rightarrow & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1 \vee D_2), \sigma(\Gamma_3), X_1 \in \sigma(D_1) \vdash_\Sigma \sigma(P_1) \in \sigma(G) & \text{I.H.} \\
\equiv & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1) \vee \sigma(D_2), \sigma(\Gamma_3), X_1 \in \sigma(D_1) \vdash_\Sigma \sigma(P_1) \in \sigma(G) & \text{Def. subst} \\
\multicolumn{2}{l}{\Gamma_1, Z \in \overline{A}, \Gamma_2, X \in D_1 \vee D_2, \Gamma_3, X_2 \in D_2 \vdash_\Sigma P_2 \in G} & \text{Ass.} \\
\Rightarrow & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1 \vee D_2), \sigma(\Gamma_3), X_2 \in \sigma(D_2) \vdash_\Sigma \sigma(P_2) \in \sigma(G) & \text{I.H.} \\
\equiv & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1) \vee \sigma(D_2), \sigma(\Gamma_3), X_2 \in \sigma(D_2) \vdash_\Sigma \sigma(P_2) \in \sigma(G) & \text{Def. subst} \\
\Rightarrow & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1) \vee \sigma(D_2), \sigma(\Gamma_3) & \\
 & \vdash_\Sigma \left(\begin{array}{ll} \texttt{case}\ X\ \texttt{of} & (\texttt{inl}\ X_1) \Rightarrow \sigma(P_1) \\ \mid & (\texttt{inr}\ X_2) \Rightarrow \sigma(P_2) \end{array} \right) \in \sigma(G) & \text{Apply } \mathsf{L}\vee \\
\equiv & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(D_1 \vee D_2), \sigma(\Gamma_3) & \\
 & \vdash_\Sigma \sigma \left(\begin{array}{ll} \texttt{case}\ X\ \texttt{of} & (\texttt{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\texttt{inr}\ X_2) \Rightarrow P_2 \end{array} \right) \in \sigma(G) & \text{Def. Subst}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, Z \in \overline{A}, \Gamma_3, X_1 \in D_1 \vdash_\Sigma P_1 \in G} & \\
\Rightarrow & \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, \sigma(\Gamma_3), X_1 \in \sigma(D_1) \vdash_\Sigma \sigma(P_1) \in \sigma(G) & \text{I.H.} \\
\Rightarrow & \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, \sigma(\Gamma_3), X_1 \in D_1 \vdash_\Sigma \sigma(P_1) \in \sigma(G) & \text{trivial} \\
\multicolumn{2}{l}{\Gamma_1, X \in D_1 \vee D_2, \Gamma_2, Z \in \overline{A}, \Gamma_3, X_2 \in D_2 \vdash_\Sigma P_2 \in G} & \\
\Rightarrow & \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, \sigma(\Gamma_3), X_2 \in \sigma(D_2) \vdash_\Sigma \sigma(P_2) \in \sigma(G) & \text{I.H.} \\
\Rightarrow & \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, \sigma(\Gamma_3), X_2 \in D_2 \vdash_\Sigma \sigma(P_2) \in \sigma(G) & \text{trivial} \\
\Rightarrow & \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, \sigma(\Gamma_3) & \\
 & \vdash_\Sigma \left(\begin{array}{ll} \texttt{case}\ X\ \texttt{of} & (\texttt{inl}\ X_1) \Rightarrow \sigma(P_1) \\ \mid & (\texttt{inr}\ X_2) \Rightarrow \sigma(P_2) \end{array} \right) \in \sigma(G) & \text{Apply } \mathsf{L}\vee \\
\equiv & \Gamma_1, X \in D_1 \vee D_2, \Gamma_2, \sigma(\Gamma_3) & \\
 & \vdash_\Sigma \sigma \left(\begin{array}{ll} \texttt{case}\ X\ \texttt{of} & (\texttt{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\texttt{inr}\ X_2) \Rightarrow P_2 \end{array} \right) \in \sigma(G) & \text{Def. subst}
\end{array}$$

Case: $\mathsf{L}\rightarrow$

$$\begin{array}{lll}
\multicolumn{2}{l}{\Gamma_1, Z \in \overline{A}, \Gamma_2, X \in G_1 \rightarrow D, \Gamma_3 \vdash_\Sigma P_1 \in G_1} & \text{Ass.} \\
\Rightarrow & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(G_1 \rightarrow D), \sigma(\Gamma_3) \vdash_\Sigma \sigma(P_1) \in \sigma(G_1) & \text{I.H.} \\
\equiv & \Gamma_1, \sigma(\Gamma_2), X \in \sigma(G_1) \rightarrow \sigma(D), \sigma(\Gamma_3) \vdash_\Sigma \sigma(P_1) \in \sigma(G_1) & \text{Def. subst}
\end{array}$$

$\Gamma_1, Z \in \overline{A}, \Gamma_2, X \in G_1 \to D, \Gamma_3, Y \in D \vdash_\Sigma P_2 \in G_2$

$\Rightarrow \ \Gamma_1, \sigma(\Gamma_2), X \in \sigma(G_1 \to D), \sigma(\Gamma_3), Y \in \sigma(D) \vdash_\Sigma \sigma(P_2) \in \sigma(G_2)$ — I.H.

$\equiv \ \Gamma_1, \sigma(\Gamma_2), X \in \sigma(G_1) \to \sigma(D), \sigma(\Gamma_3), Y \in \sigma(D) \vdash_\Sigma \sigma(P_2) \in \sigma(G_2)$ — Def. subst

$\Rightarrow \ \Gamma_1, \sigma(\Gamma_2), X \in \sigma(G_1) \to \sigma(D), \sigma(\Gamma_3)$
$\vdash_\Sigma (\texttt{let (app } X\ \sigma(P_1)) \texttt{ be } Y \texttt{ in } \sigma(P_2)) \in \sigma(G_2)$ — Apply $\mathsf{L}\to$

$\equiv \ \Gamma_1, \sigma(\Gamma_2), X \in \sigma(G_1 \to D), \sigma(\Gamma_3)$
$\vdash_\Sigma \sigma(\texttt{let (app } X\ P_1) \texttt{ be } Y \texttt{ in } P_2) \in \sigma(G_2)$ — Def. subst

$\Gamma_1, X \in G_1 \to D, \Gamma_2, Z \in \overline{A}, \Gamma_3 \vdash_\Sigma P_1 \in G_1$ — Ass.

$\Rightarrow \ \Gamma_1, X \in G_1 \to D, \Gamma_2, \sigma(\Gamma_3) \vdash_\Sigma \sigma(P_1) \in \sigma(G_1)$ — I.H.

$\equiv \ \Gamma_1, X \in G_1 \to D, \Gamma_2, \sigma(\Gamma_3) \vdash_\Sigma \sigma(P_1) \in G_1$ — triv.

$\Gamma_1, X \in G_1 \to D, \Gamma_2, Z \in \overline{A}, \Gamma_3, Y \in D \vdash_\Sigma P_2 \in G_2$ — Ass.

$\Rightarrow \ \Gamma_1, X \in G_1 \to D, \Gamma_2, \sigma(\Gamma_3), Y \in \sigma(D) \vdash_\Sigma \sigma(P_2) \in \sigma(G_2)$ — I.H.

$\equiv \ \Gamma_1, X \in G_1 \to D, \Gamma_2, \sigma(\Gamma_3), Y \in D \vdash_\Sigma \sigma(P_2) \in G_2$ — triv.

$\Rightarrow \ \Gamma_1, X \in G_1 \to D, \Gamma_2, \sigma(\Gamma_3)$
$\vdash_\Sigma (\texttt{let (app } X\ \sigma(P_1)) \texttt{ be } Y \texttt{ in } \sigma(P_2)) \in G_2$ — Apply $\mathsf{L}\to$

$\equiv \ \Gamma_1, X \in G_1 \to D, \Gamma_2, \sigma(\Gamma_3)$
$\vdash_\Sigma \sigma(\texttt{let (app } X\ P_1) \texttt{ be } Y \texttt{ in } P_2) \in G_2$ — Def. subst

Case: $\mathsf{L}\forall$

$\Gamma_1, Z \in \overline{A}, \Gamma_2, X \in \forall Y : A_G.D, \Gamma_3 \vdash_\Sigma P_1 \in \overline{A_G}$ — Ass.

$\Rightarrow \ \Gamma_1, \sigma(\Gamma_2), X \in \sigma(\forall Y : A_G.D), \sigma(\Gamma_3) \vdash_\Sigma \sigma(P_1) \in \sigma(\overline{A_G})$ — I.H.

$\equiv \ \Gamma_1, \sigma(\Gamma_2), X \in \forall Y : \sigma(A_G).\sigma(D), \sigma(\Gamma_3) \vdash_\Sigma \sigma(P_1) \in \overline{\sigma(A_G)}$ — triv.

$\Gamma_1, Z \in \overline{A}, \Gamma_2, X \in \forall Y : A_G.D, \Gamma_3, Z' \in [P_1/Y](D) \vdash_\Sigma P_2 \in G$ — Ass.

$\Rightarrow \ \Gamma_1, \sigma(\Gamma_2), X \in \sigma(\forall Y : A_G.D), \sigma(\Gamma_3), Z' \in \sigma([P_1/Y](D)) \vdash_\Sigma \sigma(P_2) \in \sigma(G)$ — I.H.

$\equiv \ \Gamma_1, \sigma(\Gamma_2), X \in \forall Y : \sigma(A_G).\sigma(D), \sigma(\Gamma_3), Z' \in [\sigma(P_1)/Y](\sigma(D))$
$\vdash_\Sigma \sigma(P_2) \in \sigma(G)$ — triv.

$\Rightarrow \ \Gamma_1, \sigma(\Gamma_2), X \in \forall Y : \sigma(A_G).\sigma(D), \sigma(\Gamma_3)$
$\vdash_\Sigma (\texttt{let (app } X\ \sigma(P_1)) \texttt{ be } Z' \texttt{ in } \sigma(P_2)) \in \sigma(G)$ — Apply $\mathsf{L}\forall$

$\Rightarrow \ \Gamma_1, \sigma(\Gamma_2), X \in \sigma(\forall Y : A_G.D), \sigma(\Gamma_3)$
$\vdash_\Sigma \sigma(\texttt{let (app } X\ P_1) \texttt{ be } Z' \texttt{ in } P_2) \in \sigma(G)$ — Def. subst

$\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, Z \in \overline{A}, \Gamma_3 \vdash_\Sigma P_1 \in \overline{A_G}$ — Ass.

$\Rightarrow \ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \sigma(\Gamma_3) \vdash_\Sigma \sigma(P_1) \in \sigma\overline{A_G}$ — I.H.

$\equiv \ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \sigma(\Gamma_3) \vdash_\Sigma \sigma(P_1) \in \overline{A_G}$ — triv.

$\Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, Z \in \overline{A}, \Gamma_3, Z' \in [P_1/Y](D) \vdash_\Sigma P_2 \in G$ — Ass.

$\Rightarrow \ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \sigma(\Gamma_3), Z' \in \sigma([P_1/Y](D)) \vdash_\Sigma \sigma(P_2) \in \sigma(G)$ — I.H.

$\equiv \ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \sigma(\Gamma_3), Z' \in [\sigma(P_1)/Y](\sigma(D)) \vdash_\Sigma$
$\sigma(P_2) \in \sigma(G)$ — Def. subst.

$\equiv \ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \sigma(\Gamma_3), Z' \in [\sigma(P_1)/Y](D)$
$\vdash_\Sigma \sigma(P_2) \in \sigma(G)$ — triv.

$\Rightarrow \ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \sigma(\Gamma_3)$
$\vdash_\Sigma (\texttt{let (app } X\ \sigma(P_1)) \texttt{ be } Z' \texttt{ in } \sigma(P_2)) \in \sigma(G)$ — Apply $\mathsf{L}\forall$

$\equiv\ \ \Gamma_1, X \in \forall Y : A_G.D, \Gamma_2, \sigma(\Gamma_3)$
$\quad \vdash_\Sigma \sigma(\texttt{let}\ (\texttt{app}\ X\ P_1)\ \texttt{be}\ Z'\ \texttt{in}\ P_2) \in \sigma(G)$ Def. subst

Case: $\mathsf{L}\exists$

$\Gamma_1, Z \in \overline{A}, \Gamma_2, X \in \exists Y : A_D.D, \Gamma_3, X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D) \vdash_\Sigma P \in G$ Ass.
$\Rightarrow\ \ \Gamma_1, \sigma(\Gamma_2), X \in \sigma(\exists Y : A_D.D), \sigma(\Gamma_3), X_1 \in \sigma\overline{A_D}, X_2 \in \sigma([X_1/Y](D))$
$\quad \vdash_\Sigma \sigma(P) \in \sigma(G)$ I.H.
$\equiv\ \ \Gamma_1, \sigma(\Gamma_2), X \in \exists Y : \sigma(A_D).\sigma(D), \sigma(\Gamma_3), X_1 \in \overline{\sigma(A_D)}, X_2 \in [\sigma(X_1)/Y](\sigma(D))$
$\quad \vdash_\Sigma \sigma(P) \in \sigma(G)$ Def. subst.
$\equiv\ \ \Gamma_1, \sigma(\Gamma_2), X \in \exists Y : \sigma(A_D).\sigma(D), \sigma(\Gamma_3), X_1 \in \overline{\sigma(A_D)}, X_2 \in [X_1/Y](\sigma(D))$
$\quad \vdash_\Sigma \sigma(P) \in \sigma(G)$ triv.
$\Rightarrow\ \ \Gamma_1, \sigma(\Gamma_2), X \in \exists Y : \sigma(A_D).\sigma(D), \sigma(\Gamma_3)$
$\quad \vdash_\Sigma (\texttt{case}\ X\ \texttt{of}\ (\texttt{inx}\ X_1\ X_2) \Rightarrow \sigma(P)) \in \sigma(G)$ Apply $\mathsf{L}\exists$
$\equiv\ \ \Gamma_1, \sigma(\Gamma_2), X \in \exists Y : \sigma(A_D).\sigma(D), \sigma(\Gamma_3)$
$\quad \vdash_\Sigma \sigma(\texttt{case}\ X\ \texttt{of}\ (\texttt{inx}\ X_1\ X_2) \Rightarrow P) \in \sigma(G)$ Def. Subst

$\Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, Z \in \overline{A}, \Gamma_3, X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D) \vdash_\Sigma P \in G$ Ass.
$\Rightarrow\ \ \Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, \sigma(\Gamma_3), X_1 \in \sigma\overline{A_D}, X_2 \in \sigma([X_1/Y](D))$
$\quad \vdash_\Sigma \sigma(P) \in \sigma(G)$ I.H.
$\equiv\ \ \Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, \sigma(\Gamma_3), X_1 \in \overline{\sigma(A_D)}, X_2 \in [\sigma(X_1)/Y](\sigma(D))$
$\quad \vdash_\Sigma \sigma(P) \in \sigma(G)$ Def. subst.
$\equiv\ \ \Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, \sigma(\Gamma_3), X_1 \in \overline{A_D}, X_2 \in [X_1/Y](D)$
$\quad \vdash_\Sigma \sigma(P) \in \sigma(G)$ triv.
$\Rightarrow\ \ \Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, \sigma(\Gamma_3)$
$\quad \vdash_\Sigma (\texttt{case}\ X\ \texttt{of}\ (\texttt{inx}\ X_1\ X_2) \Rightarrow \sigma(P)) \in \sigma(G)$ Apply $\mathsf{L}\exists$
$\equiv\ \ \Gamma_1, X \in \exists Y : A_D.D, \Gamma_2, \sigma(\Gamma_3)$
$\quad \vdash_\Sigma \sigma(\texttt{case}\ X\ \texttt{of}\ (\texttt{inx}\ X_1\ X_2) \Rightarrow P) \in \sigma(G)$ Def. subst

Case: $\mathsf{L}\Pi$

$\Gamma_1, Z \in \overline{A}, \Gamma_2, L \in \overline{\Pi x : A_G.A_D}, \Gamma_3 \vdash_\Sigma \overline{M} \in \overline{A_G}$ Ass.
$\Rightarrow\ \ \Gamma_1, \sigma(\Gamma_2), L \in \sigma\overline{\Pi x : A_G.A_D}, \sigma(\Gamma_3) \vdash_\Sigma \sigma\overline{M} \in \sigma\overline{A_G}$ I.H.
$\equiv\ \ \Gamma_1, \sigma(\Gamma_2), L \in \overline{\sigma(\Pi x : A_G.A_D)}, \sigma(\Gamma_3) \vdash_\Sigma \overline{\sigma(M)} \in \overline{\sigma(A_G)}$ Def; subst.
$\Gamma_1, Z \in \overline{A}, \Gamma_2, L \in \overline{\Pi x : A_G.A_D}, \Gamma_3, E \in \overline{((\Pi x : A_G.A_D)\ M)} \vdash_\Sigma P' \in G$ Ass.
$\Rightarrow\ \ \Gamma_1, \sigma(\Gamma_2), L \in \overline{(\Pi x : \sigma(A_G).\sigma(A_D))}, \sigma(\Gamma_3), E \in \overline{((\Pi x : \sigma(A_G).\sigma(A_D))\ \sigma(M))}$
$\quad \vdash_\Sigma \sigma(P') \in \sigma(G)$ I.H.,Def. subst
$\Rightarrow\ \ \Gamma_1, \sigma(\Gamma_2), L \in \sigma\overline{(\Pi x : A_G.A_D)}, \sigma(\Gamma_3)[\overline{(\underline{L}\ \sigma(M))}/E](\sigma(P')) \in \sigma(G)$ Apply $\mathsf{L}\Pi$
$\equiv\ \ \Gamma_1, \sigma(\Gamma_2), L \in \sigma\overline{(\Pi x : A_G.A_D)}, \sigma(\Gamma_3)\sigma([\overline{(\underline{L}\ M)}/E](P')) \in \sigma(G)$ Def subst.

alternative analog.

Case: $\mathsf{L}\Pi\mathsf{V}$ analog

$$\begin{array}{lr}
\Gamma_1, L \in \overline{\Pi x : A_G.A_D}, \Gamma_2, Z \in \overline{A}, \Gamma_3 \vdash_\Sigma \overline{M} \in \overline{A_G} & \text{Ass.} \\
\Rightarrow \quad \Gamma_1, L \in \overline{\Pi x : A_G.A_D}, \Gamma_2, \sigma(\Gamma_3) \vdash_\Sigma \overline{\sigma(M)} \in \overline{A_G} & \text{I.H., Def. subst, triv.} \\
\Gamma_1, L \in \overline{\Pi x : A_G.A_D}, \Gamma_2, Z \in \overline{A}, \Gamma_3, E \in \overline{((\Pi x : A_G.A_D)\ M)} \vdash_\Sigma P' \in G & \text{Ass.} \\
\Rightarrow \quad \Gamma_1, L \in \overline{\Pi x : A_G.A_D}, \Gamma_2, \sigma(\Gamma_3), E \in \overline{((\Pi x : A_G.A_D)\ \sigma(M))} & \\
\qquad \vdash_\Sigma \sigma(P') \in \sigma(G) & \text{I.H., Def. subst, triv.} \\
\Rightarrow \quad \Gamma_1, L \in \overline{\Pi x : A_G.A_D}, \Gamma_2, \sigma(\Gamma_3) & \\
\qquad \vdash_\Sigma \sigma([\overline{(\underline{L}\ M)}/E](P')) \in \sigma(G) & \text{Apply L}\Pi\text{, Def. subst.}
\end{array}$$

Case: L$\Pi\Sigma$

$$\begin{array}{lr}
\Gamma_1, Z \in \overline{A}, \Gamma_2 \vdash_\Sigma \overline{M} \in \overline{A_G} & \text{Ass.} \\
\Rightarrow \quad \Gamma_1, \sigma(\Gamma_2) \vdash_\Sigma \overline{\sigma(M)} \in \overline{A_G} & \text{I.H., Def. subst., triv.} \\
\Gamma_1, Z \in \overline{A}, \Gamma_2, E \in \overline{((\Pi x : A_G.A_D)\ M)} \vdash_\Sigma P' \in G & \text{Ass.} \\
\Rightarrow \quad \Gamma_1, \sigma(\Gamma_2), E \in \overline{((\Pi x : A_G.A_D)\ \sigma(M))} \vdash_\Sigma \sigma(P') \in \sigma(G) & \text{I.H., Def. subst., triv.} \\
\Rightarrow \quad \Gamma_1, \sigma(\Gamma_2) \vdash_\Sigma [\overline{(\underline{L}\ \sigma(M))}/E](\sigma(P')) \in \sigma(G) & \text{Apply L}\Pi\Sigma\text{, Def. subst.} \\
\equiv \quad \Gamma_1, \sigma(\Gamma_2) \vdash_\Sigma \sigma([\overline{(\underline{L}\ M)}/E](P')) \in \sigma(G) &
\end{array}$$

Case: L$\Pi\Sigma$V analog

Case: case Note, that $X \in \overline{B}$ occurs in Γ_1 or Γ_2: $X \in \overline{B}$ occurs left of $Z \in \overline{A}$:

$$\begin{array}{lr}
\vdash_\Sigma [P/X](\Gamma_1), Z \in \overline{A}, [P/X](\Gamma_2) \text{ ctx} & \text{by assumption} \\
\vdash_\Sigma \Gamma_1, [P/X](\Gamma_1'), \sigma([P/X](\Gamma_2)) \text{ ctx} & \text{by hyp.} \\
\Gamma_1, Z \in \overline{A}, \Gamma_2 \vdash_\Sigma P \in \overline{B} & \text{by assumption} \\
\Rightarrow \quad \Gamma_1, \sigma(\Gamma_2) \vdash_\Sigma \sigma(P) \in \overline{\sigma(B)} & \text{by hyp.} \\
\equiv \quad \sigma(\Gamma_1, \Gamma_2) \vdash_\Sigma \sigma(P) \in \overline{\sigma(B)} & \text{by def. subst.}
\end{array}$$

Therefore $[\sigma](B) = [\sigma \circ \eta](B')$, $[\sigma](\Gamma) = [\sigma \circ \eta](\Gamma')$, $[\sigma](G) = [\sigma \circ \eta](G')$. Define $\eta' = \sigma \circ \eta$. Therefore all derivations for

$$\Lambda^{(i)}, \Theta^{(i)}(\Gamma') \vdash_\Sigma P^{(i)} \in \Theta^{(i)}(G')$$

are still premisses. The application of the rule yields:

$$\begin{array}{l}
[[\sigma](P)/X]([\sigma](\Gamma_1, \Gamma_2)) \\
\vdash_\Sigma \left(\begin{array}{ll}
\mathsf{case}\ [\sigma](P)\ \mathsf{of} & \overline{c_1\ \underline{X_1^{(1)}} ... \underline{X_{m_1}^{(1)}} \Rightarrow [\sigma \circ \eta](P^{(1)})} \\
 & ... \\
\mid & \overline{c_n\ \underline{X_1^{(n)}} ... \underline{X_{m_n}^{(n)}} \Rightarrow [\sigma \circ \eta](P^{(n)})}
\end{array} \right) \in [[\sigma](P)/X]([\sigma](G))
\end{array}$$

which is equivalent to

$$\begin{array}{l}
[\sigma]([P/X](\Gamma_1, \Gamma_2)) \\
\vdash_\Sigma [\sigma] \left(\begin{array}{ll}
\mathsf{case}\ P\ \mathsf{of} & \overline{c_1\ \underline{X_1^{(1)}} ... \underline{X_{m_1}^{(1)}} \Rightarrow [\eta](P^{(1)})} \\
 & ... \\
\mid & \overline{c_n\ \underline{X_1^{(n)}} ... \underline{X_{m_n}^{(n)}} \Rightarrow [\eta](P^{(n)})}
\end{array} \right) \in [\sigma]([P/X](G))
\end{array}$$

□

Theorem 4.21 (Local reductions in MLF:) *If $\mathcal{D} :: \Gamma_1 \vdash K \in C$ and $\mathcal{E} :: \Gamma_1, Z \in C, \Gamma_2 \vdash P \in G$ then there is a derivation $\mathcal{F}$, s.t. $\mathcal{F} :: \Gamma_1, [\sigma](\Gamma_2) \vdash [\sigma](P) \in [\sigma](G)$ with $\sigma = [K/Z]$.*

Proof:

Case: Meta variables: If K is a meta variable, the four cases may occur:

Case: Let $\mathcal{D}$ be an id instantiation

$$\mathcal{D} = \frac{}{\Gamma_1, X \in C, \Gamma_2 \vdash_\Sigma X \in C}\text{id}$$

and $\mathcal{E}$ a derivation for

$$\begin{array}{c}\mathcal{E}\\ \Gamma_1, X \in C, \Gamma_2, Y \in C, \Gamma_3 \vdash_\Sigma P \in G\end{array}$$

$$\mathcal{D} \bigotimes \mathcal{E} = \begin{array}{c}\mathcal{F}\\ \Gamma_1, X \in C, \Gamma_2, [X/Y](\Gamma_3) \vdash_\Sigma [X/Y](P) \in [X/Y](G)\end{array}$$

But this derivation can be accomplished by contraction lemma 4.17.

Case: Let $\mathcal{D}$ be a derivation for

$$\begin{array}{c}\mathcal{D}\\ \Gamma_1 \vdash_\Sigma P \in C\end{array}$$

and $\mathcal{E}$ be a derivation for

$$\mathcal{E} = \frac{}{\Gamma_1, X \in C, \Gamma_2 \vdash_\Sigma X \in C}\text{id}$$

$$\mathcal{D} \bigotimes \mathcal{E} = \begin{array}{c}\mathcal{F}\\ \Gamma_1, [P/X](\Gamma_2) \vdash_\Sigma P \in C\end{array}$$

but this follows directly from the weakening lemma 4.16.

Case: Let $\mathcal{D}$ be an const instantiation, $c : A$ a signature entry in Σ,

$$\mathcal{D} = \frac{}{\Gamma_1 \vdash_\Sigma \overline{c} \in \overline{A}}\text{const}$$

and $\mathcal{E}$ a derivation for

$$\begin{array}{c}\mathcal{E}\\ \Gamma_1, X \in \overline{A}, \Gamma_2 \vdash_\Sigma P \in G\end{array}$$

$$\mathcal{D} \bigotimes \mathcal{E} = \begin{array}{c}\mathcal{F}\\ \Gamma_1, [\overline{c}/X](\Gamma_2) \vdash_\Sigma [\overline{c}/X](P) \in [\overline{c}/X](G)\end{array}$$

but by the substitution lemma 4.19 we can derive the same formula from $\mathcal{E}$ without using cut.

Case: Let $\mathcal{D}$ be a derivation of

$$\overset{\mathcal{D}}{\Gamma_1 \vdash_\Sigma P \in C}$$

and $\mathcal{E}$ an **const** instantiation, $c : A$ a signature entry in Σ,

$$\mathcal{E} = \frac{}{\Gamma_1, X \in C, \Gamma_2 \vdash_\Sigma \overline{c} \in \overline{A}}\,\mathsf{const}$$

$$\mathcal{D} \bigotimes \mathcal{E} = \overset{\mathcal{F}}{\Gamma_1, [\overline{c}/X](\Gamma_2) \vdash_\Sigma \overline{c} \in \overline{A}}$$

but this follows directly from the weakening lemma 4.16.

Case: `fun` -programs

Case: Universal quantification:Let $\mathcal{D}$ be a derivation for

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1}{\Gamma_1, Y \in \overline{A_P} \vdash_\Sigma [Y/X]Q \in [Y/X](C)}}{\Gamma_1 \vdash_\Sigma (\mathtt{fun}\,X.Q) \in \forall X : A_P.C}\,\mathsf{R}\forall$$

and

$$\mathcal{E} = \frac{\overset{\mathcal{E}_1}{\Gamma_1, F \in \forall X : A_P.C, \Gamma_2 \vdash_\Sigma P \in \overline{A_P}} \qquad \overset{\mathcal{E}_2}{\Gamma_1, F \in \forall X : A_P.C, \Gamma_2, E \in [P/X](C) \vdash_\Sigma P' \in G}}{\Gamma_1, F \in \forall X : A_P.G, \Gamma_2 \vdash_\Sigma (\mathtt{let}\ (\mathsf{app}\ F\ P)\ \mathtt{be}\ E\ \mathtt{in}\ P') \in G}\,\mathsf{L}\forall$$

$$F \notin Free(A) \quad \text{since } Free(A_P) \subset sup(\Gamma_1) \text{ and } F \notin \Gamma_1 \tag{C.1}$$
$$F \notin Free(C) \quad \text{since for } Y \text{ new, } Free([Y/X](C)) \setminus \{Y\} \subset sup(\Gamma_1) \tag{C.2}$$

Applying the cut rule to the derivations $\mathcal{D}$ and $\mathcal{E}$ we obtain

$$\mathcal{D} \bigotimes \mathcal{E} = \overset{\mathcal{F}}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](\mathtt{let}\ (\mathsf{app}\ F\ P)\ \mathtt{be}\ E\ \mathtt{in}\ P') \in [\sigma](G)}$$

where we use σ as an abbreviation for the substitution $[(\mathtt{fun}\,X.Q)/F]$

$$\mathcal{D} \bigotimes \mathcal{E}_1 = \overset{\mathcal{F}_1}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](P) \in [\sigma]\overline{A_P}}$$

Because of (C.1) this is equivalent to

$$\mathcal{D} \bigotimes \mathcal{E}_1 = \overset{\mathcal{F}_1}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](P) \in \overline{A_P}}$$

$$\mathcal{D} \bigotimes \mathcal{E}_2 = \overset{\mathcal{F}_2}{\Gamma_1, [\sigma](\Gamma_2), E \in [\sigma]([P/X](C)) \vdash_\Sigma [\sigma](P') \in [\sigma](G)}$$

Since in Γ_1 $[\sigma]([P/X](C)) = [[\sigma](P)/Y]([\sigma]([Y/X](C)))$ and (C.2) this equation simplifies to

$$\mathcal{D} \bigotimes \mathcal{E}_2 = \quad \begin{array}{c} \mathcal{F}_2 \\ \Gamma_1, [\sigma](\Gamma_2), E \in [[\sigma](P)/X](C) \vdash_\Sigma [\sigma](P') \in [\sigma](G) \end{array}$$

To cut $\mathcal{F}_1$ with $\mathcal{D}_1$ we have to weaken $\mathcal{D}_1$ first: $\mathcal{D}_1' = [\Gamma_1 \bigvee [\sigma](\Gamma_2)]\mathcal{D}_1$ which exists by lemma 4.16:

$$\begin{array}{c} \mathcal{D}_1' \\ \Gamma_1, [\sigma](\Gamma_2), Y \in \overline{A_P} \vdash_\Sigma [Y/X]Q \in [Y/X](C) \end{array}$$

$$\mathcal{D}_1' \bigotimes \mathcal{F}_1 = \quad \begin{array}{c} \mathcal{F}_3 \\ \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [[\sigma](P)/Y]([Y/X](Q)) \in [[\sigma](P)/Y]([Y/X](C)) \end{array}$$

which simplifies to

$$\mathcal{D}_1' \bigotimes \mathcal{F}_1 = \quad \begin{array}{c} \mathcal{F}_3 \\ \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [[\sigma](P)/X](Q) \in [[\sigma](P)/X](C) \end{array}$$

$$\mathcal{F}_2 \bigotimes \mathcal{F}_3 = \quad \begin{array}{c} \mathcal{F}_4 \\ \Gamma_1, [\sigma](\Gamma_2) \ \vdash_\Sigma [[[\sigma](P)/X](Q)/E]([\sigma](P')) \in [[[\sigma](P)/X](Q)/E]([\sigma](G)) \end{array}$$

and trivially:

$$\mathcal{F}_2 \bigotimes \mathcal{F}_3 = \quad \begin{array}{c} \mathcal{F}_4 \\ \Gamma_1, [\sigma](\Gamma_2) \ \vdash_\Sigma [[[\sigma](P)/X](Q)/E]([\sigma](P')) \in [\sigma](G) \end{array}$$

Case: Implication

$$\mathcal{D} = \dfrac{\begin{array}{c} \mathcal{D}_1 \\ \Gamma_1, X \in C_1 \vdash_\Sigma Q \in C_2 \end{array}}{\Gamma_1 \vdash_\Sigma (\mathtt{fun}\, X.Q) \in C_1 \to C_2} \,\mathsf{R}\!\to$$

$$\mathcal{E} = \dfrac{\begin{array}{c} \mathcal{E}_1 \\ \Gamma_1, F \in C_1 \to C_2, \Gamma_2 \vdash_\Sigma P \in C_1 \end{array} \qquad \begin{array}{c} \mathcal{E}_2 \\ \Gamma_1, F \in C_1 \to C_2, \Gamma_2, E \in C_2 \vdash_\Sigma P' \in G \end{array}}{\Gamma_1, F \in C_1 \to C_2, \Gamma_2 \vdash_\Sigma (\mathtt{let}\ (\mathtt{app}\ F\ P)\ \mathtt{be}\ E\ \mathtt{in}\ P') \in G} \,\mathsf{L}\!\to$$

$$\mathcal{D} \bigotimes \mathcal{E} = \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](\mathtt{let}\ (\mathtt{app}\ F\ P)\ \mathtt{be}\ E\ \mathtt{in}\ P') \in [\sigma](G)$$

with $[\sigma] = [(\mathtt{fun}\, X.Q)/F]$. Apply Cut to $\mathcal{D}$ and $\mathcal{E}_1$:

$$\mathcal{D} \bigotimes \mathcal{E}_1 = \quad \begin{array}{c} \mathcal{F}_1 \\ \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](P) \in [\sigma](C_1) \end{array}$$

which is equivalent to

$$\begin{array}{c} \mathcal{F}_1 \\ \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](P) \in C_1 \end{array}$$

Apply Cut to $\mathcal{D}$ and $\mathcal{E}_1$:

$$\mathcal{D}\bigotimes\mathcal{E}_2 = \overset{\mathcal{F}_2}{\Gamma_1, [\sigma](\Gamma_2), E \in [\sigma](C_2) \vdash_\Sigma [\sigma](P') \in [\sigma](G)}$$

which is equivalent to

$$\overset{\mathcal{F}_2}{\Gamma_1, [\sigma](\Gamma_2), E \in C_2 \vdash_\Sigma [\sigma](P') \in [\sigma](G)}$$

Weaken $\mathcal{D}_1$ to $\mathcal{D}_1'$ as $\mathcal{D}_1[\Gamma_1 \bigvee [\sigma](\Gamma_2)]$:

$$\overset{\mathcal{D}_1'}{\Gamma_1, [\sigma](\Gamma_2), X \in C_1 \vdash_\Sigma Q \in C_2}$$

Apply Cut to $\mathcal{F}_1$ and $\mathcal{D}_1'$:

$$\mathcal{F}_1\bigotimes\mathcal{D}_1' = \overset{\mathcal{F}_3}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [[\sigma](P)/X](Q) \in [[\sigma](P)/X](C_2)}$$

which is equivalent to

$$\overset{\mathcal{F}_3}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [[\sigma](P)/X](Q) \in C_2}$$

Apply Cut to $\mathcal{F}_3$ and $\mathcal{F}_2$:

$$\mathcal{F}_3\bigotimes\mathcal{F}_2 = \overset{\mathcal{F}_4}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [[[\sigma](P)/X](Q)/E]([\sigma](P')) \in [[[\sigma](P)/X](Q)/E]([\sigma](G))}$$

which is equivalent to

$$\overset{\mathcal{F}_4}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [[[\sigma](P)/X](Q)/E]([\sigma](P')) \in [\sigma](G)}$$

Case: inx- programs

$$\dfrac{\overset{\mathcal{D}_1}{\Gamma_1 \vdash_\Sigma P' \in \overline{A_P}} \qquad \overset{\mathcal{D}_2}{\Gamma_1 \vdash_\Sigma P \in [P'/X](C)}}{\mathcal{D} = \Gamma_1 \vdash_\Sigma (\mathbf{inx}\ P'\ P) \in \exists X : A_P.C}\,\mathsf{R}\exists$$

$$\dfrac{\overset{\mathcal{E}_1}{\Gamma_1, X \in \exists Y : A_P.C, \Gamma_2, X_1 \in \overline{A_P}, X_2 \in [X_1/Y](C) \vdash_\Sigma Q \in G}}{\mathcal{E} = \Gamma_1, X \in \exists Y : A_P.C, \Gamma_2 \vdash_\Sigma (\mathsf{case}\ X\ \mathsf{of}\ (\mathbf{inx}\ X_1\ X_2) \Rightarrow Q) \in G}\,\mathsf{L}\exists$$

$$\mathcal{D}\bigotimes\mathcal{E} = \overset{\mathcal{F}}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](\mathsf{case}\ X\ \mathsf{of}\ (\mathbf{inx}\ X_1\ X_2) \Rightarrow Q) \in [\sigma](G)}$$

with $[\sigma] = [(\mathbf{inx}\, P'\ P)/X]$.

$$\mathcal{D} \bigotimes \mathcal{E}_1 = \overset{\mathcal{F}_1}{\Gamma_1, [\sigma](\Gamma_2), X_1 \in [\sigma](\overline{A_P}), X_2 \in [\sigma]([X_1/Y](C)) \vdash_\Sigma [\sigma](Q) \in [\sigma](G)}$$

which is trivially equivalent to

$$\mathcal{D} \bigotimes \mathcal{E}_1 = \overset{\mathcal{F}_1}{\Gamma_1, [\sigma](\Gamma_2), X_1 \in (\overline{A_P}), X_2 \in [X_1/Y](C) \vdash_\Sigma [\sigma](Q) \in [\sigma](G)}$$

As above we have to weaken $\mathcal{D}_1$ and $\mathcal{D}_2$.

$$\overset{\mathcal{D}_1'}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma P' \in (\overline{A_P})} \qquad \overset{\mathcal{D}_2'}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma P \in [P'/Y](C)}$$

We can cut $\mathcal{D}_1'$ and $\mathcal{F}_1$ to obtain:

$$\mathcal{D}_1' \bigotimes \mathcal{F}_1 = \overset{\mathcal{F}_2}{\Gamma_1, [\sigma](\Gamma_2), X_2 \in [P'/X_1]([X_1/Y](C)) \vdash_\Sigma [P'/X_1]([\sigma](Q)) \in [P'/X_1]([\sigma](G))}$$

Trivially this is equivalent to

$$\overset{\mathcal{F}_2}{\Gamma_1, [\sigma](\Gamma_2), X_2 \in [P'/Y](C) \vdash_\Sigma [P'/X_1]([\sigma](Q)) \in [\sigma](G)}$$

Finally the application of cut to $\mathcal{D}_2$ and $\mathcal{F}_2$ yields:

$$\mathcal{D}_2' \bigotimes \mathcal{F}_2 = \overset{\mathcal{F}_3}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [P/X_2]([P'/X_1]([\sigma](Q))) \in [P/X_2]([\sigma](G))}$$

which is again equivalent to

$$\overset{\mathcal{F}_3}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [P/X_2]([P'/X_1]([\sigma](Q))) \in [\sigma](G)}$$

Case: inl- programs

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\Gamma_1 \vdash_\Sigma P \in C_1}}{\Gamma_1 \vdash_\Sigma (\mathbf{inl}\ P) \in C_1 \vee C_2}\,\mathsf{R}\vee_1$$

$$\mathcal{E} = \cfrac{\overset{\mathcal{E}_1}{\Gamma_1, X \in C_1 \vee C_2, \Gamma_2, X_1 \in C_1 \vdash_\Sigma P_1 \in G} \qquad \overset{\mathcal{E}_2}{\Gamma_1, X \in C_1 \vee C_2, \Gamma_2, X_2 \in C_2 \vdash_\Sigma P_2 \in G}}{\Gamma_1, X \in C_1 \vee C_2, \Gamma_2 \vdash_\Sigma \begin{pmatrix} \mathtt{case}\ X\ \mathtt{of} & (\mathbf{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\mathbf{inr}\ X_2) \Rightarrow P_2 \end{pmatrix} \in G}\,\mathsf{L}\vee$$

$$\mathcal{D} \bigotimes \mathcal{E} = \overset{\mathcal{F}}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma] \begin{pmatrix} \mathtt{case}\ X\ \mathtt{of} & (\mathbf{inl}\ X_1) \Rightarrow P_1 \\ \mid & (\mathbf{inr}\ X_2) \Rightarrow P_2 \end{pmatrix} \in [\sigma](G)}$$

where $[\sigma] = [(\texttt{inl}\ P)/X]$. Apply Cut to $\mathcal{D}$ and $\mathcal{E}_1$:

$$\mathcal{D} \bigotimes \mathcal{E}_1 = \quad \overset{\mathcal{F}_1}{\Gamma_1, [\sigma](\Gamma_2), X_1 \in [\sigma](C_1) \vdash_\Sigma [\sigma](P_1) \in [\sigma](G)}$$

which is equivalent to

$$\overset{\mathcal{F}_1}{\Gamma_1, [\sigma](\Gamma_2), X_1 \in C_1 \vdash_\Sigma [\sigma](P_1) \in [\sigma](G)}$$

Weaken $\mathcal{D}_1$ to $\mathcal{D}'_1$ with $\mathcal{D}_1[\Gamma_1 \bigvee [\sigma](\Gamma_2)]$:

$$\overset{\mathcal{D}'_1}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma P \in C_1}$$

Apply Cut to $\mathcal{D}'_1$ and $\mathcal{F}_1$:

$$\mathcal{D}'_1 \bigotimes \mathcal{F}_1 = \quad \overset{\mathcal{F}_2}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [P/X_1]([\sigma](P_1)) \in [P/X_1]([\sigma](G))}$$

which is equivalent to

$$\mathcal{D}'_1 \bigotimes \mathcal{F}_1 = \quad \overset{\mathcal{F}_2}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [P/X_1]([\sigma](P_1)) \in [\sigma](G)}$$

Case: inr-programs

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma_1 \vdash_\Sigma P \in C_2}}{\Gamma_1 \vdash_\Sigma (\texttt{inl}\ P) \in C_1 \vee C_2}\,\mathsf{R}\vee_2$$

goes analog to previous case

Case: pair-programs

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma_1 \vdash_\Sigma P_1 \in C_1} \qquad \overset{\mathcal{D}_2}{\Gamma_1 \vdash_\Sigma P_2 \in C_2}}{\Gamma_1 \vdash_\Sigma (\mathsf{pair}\ P_1\ P_2) \in C_1 \wedge C_2}\,\mathsf{R}\wedge$$

and

$$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma_1, X \in C_1 \wedge C_2, \Gamma_2, X_1 \in C_1, X_2 \in C_2 \vdash_\Sigma P \in G}}{\Gamma_1, X \in C_1 \wedge C_2, \Gamma_2 \vdash_\Sigma \big(\ \mathsf{case}\ X\ \mathsf{of}\quad (\mathsf{pair}\ X_1\ X_2) \Rightarrow P\ \big) \in G}\,\mathsf{L}\wedge$$

$$X \notin \mathit{Free}(C_1) \cup \mathit{Free}(C_2), \quad \text{since } X \notin \mathit{sup}(\Gamma_1) \tag{C.3}$$

$$\tag{C.4}$$

$$\mathcal{D} \bigotimes \mathcal{E} = \quad \overset{\mathcal{F}}{\Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [\sigma](\mathsf{case}\ X\ \mathsf{of}\ (\mathsf{pair}\ X_1\ X_2) \Rightarrow P) \in [\sigma](G)}$$

where σ is a shorthand for $[(\texttt{pair}\ P_1\ P_2)/X]$

$$\mathcal{D} \bigotimes \mathcal{E}_1 = \quad \begin{array}{c} \mathcal{F}_1 \\ \Gamma_1, [[\sigma]](\Gamma_2), X_1 \in [\sigma](C_1), X_2 \in [\sigma](C_2) \vdash_\Sigma [\sigma](P) \in [\sigma](G) \end{array}$$

This is because of (C.3) equivalent to

$$\mathcal{D} \bigotimes \mathcal{E}_1 = \begin{array}{c} \mathcal{F}_1 \\ \Gamma_1, [\sigma](\Gamma_2), X_1 \in C_1, X_2 \in C_2 \vdash_\Sigma [\sigma](P) \in [\sigma](G) \end{array}$$

We have to weaken $\mathcal{D}_1, \mathcal{D}_2$, to perform cut elimination with $\mathcal{F}_1$ — apply weakening lemma 4.16.

$$\begin{array}{c} \mathcal{D}'_1 \\ \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma P_1 \in C_1 \end{array}$$

$$\begin{array}{c} \mathcal{D}'_2 \\ \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma P_2 \in C_2 \end{array}$$

$$\mathcal{D}'_1 \bigotimes \mathcal{F}_1 = \quad \begin{array}{c} \mathcal{F}_2 \\ \Gamma_1, [\sigma](\Gamma_2), X_2 \in [P_1/X_1](C_2) \vdash_\Sigma [P_1/X_1]([\sigma](P)) \in [P_1/X_1]([\sigma](G)) \end{array}$$

Trivially this is equivalent to

$$\mathcal{D}'_1 \bigotimes \mathcal{F}_1 = \quad \begin{array}{c} \mathcal{F}_2 \\ \Gamma_1, [\sigma](\Gamma_2), X_2 \in C_2 \vdash_\Sigma [P_1/X_1]([\sigma](P)) \in [\sigma](G) \end{array}$$

Finally cut elimination gives us

$$\mathcal{D}'_2 \bigotimes \mathcal{F}_2 = \quad \begin{array}{c} \mathcal{F}_3 \\ \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [P_2/X_2]([P_1/X_1]([\sigma](P))) \in [P_2/X_2]([\sigma](G)) \end{array}$$

which is equivalent to

$$\mathcal{D}'_2 \bigotimes \mathcal{F}_2 = \quad \begin{array}{c} \mathcal{F}_3 \\ \Gamma_1, [\sigma](\Gamma_2) \vdash_\Sigma [P_2/X_2]([P_1/X_1]([\sigma](P))) \in [\sigma](G) \end{array}$$

□

Bibliography

[AM82] U. Montenari A. Martelli. An efficient unification algorithm. In *ACM TOPLAS*, volume 4/2, pages 258–282, April 1982.

[Bar92] Henk Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of logic in Computer Science*, volume II, pages 118–309. Oxford University Press, 1992.

[C$^+$86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[C$^+$95] Christina Cornes et al. *The Coq Proof Assistant, Reference Manual, Version 5.10.* INRIA, CNRS, France, 1995.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[dB72] N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formulamanipulation with application to the church-rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in Coq. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.

[Gal93] Jean Gallier. Constructive logics Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110:249–339, 1993.

[GLT88] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1988.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

[Hue88] Gérard Huet. Induction principles formalized in the calculus of constructions. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers.* Elsevier Science, 1988.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[LPM94] Francois Leclerc and Christine Paulin-Mohring. Programming with streams in Coq: A case study: the sieve of eratosthenes. In *TYPES '93*, number 383 in LNCS, pages 191–212, 1994.

[Mag93] Lena Magnusson. Refinement and local undo in the interactive proof editor ALF. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 191–208, Nijmegen, The Netherlands, 1993.

[Mag95] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[ML84] Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory Lecture Notes. BIBLIOPOLIS, 1984.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[ORS92] Sam Owre, John Rusby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Conference on Auotmated deduction (CADE)*, volume 607 of *LNCS*, pages 748–752, Saratoga, NY, 1992.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[Pfe99] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 199? To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe92] Frank Pfenning. Computation and deduction. Unpublished lecture notes, revised May 1994, May 1992.

[Pfe94a] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe94b] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Pfe94c] Frank Pfenning. A structural proof of cut elimination and its representation in a logical framework. Technical Report CMU-CS-94-218, Department of Computer Science, Carnegie Mellon University, November 1994.

[Pfe95] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[PM93] Christine Paulin-Mohring. Inductive definitinos in the system Coq — rules and properties. In M. Bezem and J.-F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in LNCS, 1993.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[RSC95] J. Rushby and D.W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, SRI International, 95.

[Sny91] Wayne Snyder. *A proof Theory for general Unification.* Birkhäuser, 1991.